Interim Scientific Report 2

TECHNIQUES FOR THE REALIZATION OF ULTRARELIABLE SPACEBORNE COMPUTERS

By: J. GOLDBERG M. W. GREEN K. N. LEVITT H. S. STONE

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
ELECTRONICS RESEARCH CENTER
575 TECHNOLOGY SQUARE
CAMBRIDGE, MASSACHUSETTS 02139

CONTRACT NAS12-33

STANFORD RESEARCH INSTITUTE

MENLO PARK, CALIFORNIA

STANFORD RESEARCH INSTITUTE
MENLO PARK, CALIFORNIA

October 1967

Interim Scientific Report 2

TECHNIQUES FOR THE REALIZATION OF ULTRARELIABLE SPACEBORNE COMPUTERS

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
ELECTRONICS RESEARCH CENTER
575 TECHNOLOGY SQUARE
CAMBRIDGE, MASSACHUSETTS 02139

CONTRACT NAS12-33

By: J. GOLDBERG M. W. GREEN K. N. LEVITT H. S. STONE

SRI Project 5580

Approved: D. R. BROWN, MANAGER
COMPUTER TECHNIQUES LABORATORY

J. D. NOE, EXECUTIVE DIRECTOR
INFORMATION SCIENCE AND ENGINEERING

Copy No. 12

ABSTRACT

This is the second scientific report of a study of the development techniques for the realization of ultrareliable, high-performance, spaceborne computers. The techniques developed are in support of computer structures in which reliability is achieved through autonomously controlled logical reconfiguration and fault masking. A multiprocessor model is described that is particularly appropriate to the attainment of ultrareliability. Local design techniques, which facilitate reconfiguration, are discussed for various computer functions, including memory and microprogram control. Design techniques are presented for economical, fault-tolerant, data commutation networks. An initial effort is being directed toward a formal description of program-design techniques that will facilitate hardware diagnosis and, hopefully, yield mistake-free programs.

FOREWORD

This is an interim report, summarizing work accomplished during the first six months of the second phase of a two year program, the goal of which is the development of techniques for the realization of ultra-reliable space computers. This study has been conducted in the Computer Techniques Laboratory of Stanford Research Institute, under the sponsorship of the Electronics Research Center of the National Aeronautics and Space Administration.

The goals of the first phase were to survey the state of the art of design for achieving ultrareliable spaceborne computers, and to form a basis for research which would advance that art. The final report, which resulted from the first phase of the program, was concerned with the following:

(1) The basic characteristics of an advanced spaceborne computer

(2) A description of fault-masking techniques for general logic functions

(3) A survey of codes for storage and arithmetic operations

(4) Problems of system organization for dynamic error control

(5) Tests for diagnosis of fault conditions

(6) Some initial descriptions of network designs for a reconfigurable computer, including commutation or interconnection networks, programmable processing modules and programmable control units

(7) Error-control techniques for memory systems

(8) Distributed power supply systems

(9) The application of magnetic logic

(10) A survey of the published literature on the attainment of reliable systems through the use of redundancy.

The goal of the second phase is to develop detailed techniques for the logical design of an advanced, ultrareliable spaceborne computer. The techniques to be developed are to be used in support of computer structures in which reliability is achieved through autonomously-controlled logical reconfiguration and fault masking. In particular, those techniques have been developed by following a certain method of approach, which entails certain steps. The first step in the approach involves the development of a system organization that facilitates dynamic maintenance processes. On the basis of the selected system organization, a detailed logical design is then performed of networks that are uniquely appropriate for a reconfigurable computer. Thirdly, diagnostic procedures, reliability enhancement techniques and reliability analysis measures are developed for these networks, where the requirement exists. The next step in the approach requires that software techniques be developed to aid in the diagnosis and detection of failures. Also, techniques must be developed for designing reliable programs. Finally, reliability analysis techniques are developed for the overall system.

The report is organized into six chapters and one Appendix. The first chapter which serves as an overall introduction to the report, contains the statement of the problem; the goals, methods, and assumptions of the study; and a brief review of prior work on pertinent aspects of reliability enhancement; in addition to the organization of the report.

Chapter II is concerned with the principles of multiprocessor system design that are particularly appropriate to the attainment of ultra-reliability. Chapter III contains logical design techniques for networks identified with the memory, control, and microprogram control functions. Chapter IV contains design details for commutation networks that are to perform the important function of data switching in a multiprocessor computer. Chapter V contains a formal description of program-design techniques, which will facilitate hardware diagnosis and which will, hopefully, yield mistake-free programs. Chapter VI contains a brief

description of other topics considered in the study, namely network diagnosis, and a survey of the pertinent literature, and also the conclusions and a summary of our plans for the remainder of the program.

The report is self-contained at least as far as the statement of principles is concerned. Detailed mathematical proofs and the description of some hardware designs, particularly concerning commutation networks and arithmetic processing elements have been deferred until publication of the final report. The reader is referred to the first phase final report for detailed background information.

The technical studies reported herein are the work of the following members of the Computer Techniques Laboratory:

Mr. J. Goldberg

Mr. M. W. Green

Professor E. L. Lawler (University of Michigan-Summer employee of SRI)

Dr. K. N. Levitt

Dr. R. A. Short

Dr. H. S. Stone

Dr. J. B. Turner

All of the individuals contributed to the writing of the various sections of the report. Mr. Goldberg who serves as Project Supervisor and Dr. K. N. Levitt who serves as Project Leader are responsible for the organization and editing of the report.

CONTENTS

CONTENTS (Continued)

CONTENTS (Concluded)

ILLUSTRATIONS

ILLUSTRATIONS (Concluded)

TABLES

I INTRODUCTION

In this chapter we discuss briefly and in general terms, the problem of realizing ultrareliable spaceborne computers. Specifically, the chapter contains a discussion of the problem of designing computers that are appropriate to the characteristics of space missions, the statement of the program goals, a brief review of prior work on the attainment of ultrareliability, and a brief summary of this report.

A. Statement of the Problem

In this section we consider the basic characteristics of space mission computation, which impose severe constraints on the spaceborne computer design; and the conclusions, concerning the design, that are a consequence of these constraints.* The advanced spaceborne computer must respond to the following.

1. Multiple-Problem Sets

Several problems must be accommodated simultaneously, implying that a multiprocessing and/or a multiprogramming capability is required.

2. Highly-Varied and Complex Computations

It is anticipated that the scope of the mission will require a general scientific-type computer that will accommodate to a wide variety of sensors and output devices.

3. Computations with a Range of Priorities

It is convenient to assign to each mission computation three priority measures. The first of these is a critical value, which indicates the relative need for the existence of a particular computation, compared with the other computations at the moment. For example, certain launch

---

* References are listed at the end of the report.

computations are probably essential compared with certain ion-density computations. The second such measure is an accuracy value, reflecting the value attached to varying degrees of accuracy in the particular computation. For example, although a 2000-mile pass may be desired, a 20,000-mile pass may still be tolerable if the closer pass simply cannot be attained. The third measure is an urgency value, which reflects the required speed with which a certain computation must be performed and hence, the amount of equipment that must be devoted to its execution. These characteristics imply one attractive approach. A computer may be designed which is capable of altering the logical interconnections among the computer components and, the tasks may be scheduled to match the available performance capability. Such an organization is, colloquially, said to embody reconfiguration with graceful degradation.

4. Variable Capacity of the Earth-Vehicle Communication Link

During certain phases of the mission, communication to the vehicle will not be possible, although there will be many instances when high speed communication--of a rate possibly exceeding 1 megabit/sec--will be quite feasible. Hence, the computer must be capable of autonomously carrying out diagnosis and repair routines during part of the mission, and conversely must be capable of responding to (and indeed taking advantage of) external control information during other times.

5. Failures Which May Be Transient or Permanent

This fact is, of course, obvious, but the optimum technique, which will distinguish between these two fault types, is not at all obvious. One approach is to treat all failures as transient and effect a try-again procedure in response to all failures.

6. Failures Which May Not Be Independent or May Not Embrace Single Components

It is clear that the potentially high-stress space environment can result in a failure that will not be confined to a single component. For example, a radiation pulse could affect a sizable portion of the system, and a sudden unexpected acceleration could result in a fractured

chip. It has been convenient in the past to assume that failures (possibly embracing many elements as described above) occur one at a time. This assumption might be tenable if each failed subsystem is repaired immediately following the fault occurrence. One possible counterexample is the case wherein a system relies upon the switching of standby units to achieve ultra-reliability. It is required that such standby units either possess fault detection capabilities, or that they are diagnosed immediately prior to insertion in the system.

7. Severe Constraints on Weight and Power

Although the constraints on the weight and power of spaceborne equipment have been relaxed in recent years, it remains imperative to achieve a design which provides the maximum ratio of computing power per total equipment. This observation provides additional evidence for a multiprocessing-graceful degradation system, with a minimum amount of pure standby redundancy.

8. Failures in Most Components

Statistical reliability measures, comprehending various risk policies, have been shown to provide useful estimates of system performance. However, since the variance of most semiconductor failure distributions is quite large, it is important to minimize the number of blocks for which a single component failure would disable the entire system. Essentially, it is important to incorporate redundancy into as much of the system as possible, even though, on a statistical basis, not much improvement is realized by the inclusion of such redundancy.*

B. Statement of Program Goals

In recognition of the severe problem of designing ultrareliable spaceborne computers, NASA Electronics Research Center has set up the following study goals, the first three of which relate to the completed

---

* It is apparent that the design philosophy should reflect Murphy's Law.

first phase, and the latter four of which relate to the half-completed second phase:

(1) To survey the state of the art of logical design of spaceborne computers as it pertains to the enhancement of reliability

(2) To conceive and evaluate new schemes of system design and operation that offer promise of advancing the state of the art

(3) To recommend further directions of research that will aid in the improvement of present techniques

(4) To investigate the organization of aerospace computers having high computational performance in which autonomously controlled logical reconfiguration of equipment and fault masking are employed for the purpose of achieving ultrareliable operation. Derive specifications for classes of networks and processes that are appropriate to the realization of such organizations

(5) To investigate the design of networks that realize the functions of general logic and commutation for reconfiguration for the systems derived in Item 4. These networks will incorporate various features of error control including fault masking, diagnosability, and reconfigurability. The design will employ criteria appropriate to advanced integrated-semiconductor array technology. Develop logical designs and techniques of analysis and synthesis appropriate to the particular networks designed. Develop criteria for the evaluation of the reliability of such networks.

(6) To investigate the design of programs that facilitate the reconfiguration processes in systems of the type developed in Item 4 and that facilitate the flexible variation of computational performance with amount of available equipment. Propose the specification of requirements on executive programs that consider reconfiguration. Develop schemes for re-addressing and replacement of hardware control by software subroutines.

(7) To investigate techniques for the evaluation of the reconfigurable computers proposed in Item 4. Develop approaches to the construction of theoretical reliability models for such computers.

C. Brief Review of Prior Work

A significant effort has been devoted in the past decade toward solving various facets of the problem of realizing ultrareliable digital systems. This work, summarized in detail in the Phase I--Final Report,[1]

ranges from investigations of fault-masking techniques for various computer sub-blocks, such as arithmetic processor and memory units, to studies of reliability-enhancement policies for large systems. It is felt that most of the problems pertaining to enhancing the reliability of isolated digital sub-blocks are now understood, at least from the standpoint of making sound engineering judgments concerning the use of the various techniques.* Strictly passive redundancy techniques have been applied to the control and arithmetic processing sections of the Saturn IVb guidance computer, and it has been concluded[2] that the application of such techniques, exclusively, cannot economically satisfy the computation and reliability requirements of future spaceborne computers.

The anticipation of this conclusion prompted an increased effort on the part of many organizations in the investigation of dynamic error-control mechanisms, in which the logical interconnections among the components of the computer may be altered. Theoretically this approach enables more efficient utilization of redundancy than the passive approach. In an attempt to substantiate this supposition, Avizienis,[3,4] et al. have been working on the construction of the JPL-STAR reconfigurable guidance computer, which embodies probably the simplest form of reconfiguration. This computer consists of a set of identical processing units, each with self-contained error-detection control by means of an arithmetic code, and a reliable magnetic power stepping switch,† which can connect power to any one of the units. IBM[5] is presently building a reconfigurable version of the Saturn IVb guidance computer. In these systems the reconfiguration is employed only at very high functional levels but it is well-known that there is potentially greater gain to be achieved by employing the reconfiguration at lower system levels.

---

* Three of the most promising so-called passive techniques discussed in Ref. 1 are the use of replicated-voting logic, error-correcting codes, and adaptive voting logic.

† In a forthcoming SRI report the operation of this power switch will be described.

Graceful-Degradation Systems which enable more efficient utilization of available equipment have been conceived and supposedly evaluated. Among the many references on this approach, in Ref. 6 a single processor structure is assumed, and in Refs. 7 and 8 a multiprocessor structure is postulated. In these studies as well as many others, several important items are not treated in depth, namely those relating to the following:

(1) Diagnostic and replacement policies

(2) Logical design techniques for memory, control, and processing units so that diagnosis and repair are facilitated (or indeed feasible)

(3) Reliable commutation (or data switching) required for the execution of subsystem replacement

(4) The specification of software for the control of diagnosis and repair.

When the new sources of failure that are introduced by the mechanization of the four items are included in the reliability analysis, it is not clear that the systems will perform as promised. This is an especially intrical problem when reconfigurability is extended to low-system levels, or when the capability for graceful degradation is provided.

D. Brief Summary of Report

The succeeding chapters represent an attempt to study in depth many of the detailed problems which must be investigated before a design for a reconfigurable spaceborne computer with graceful degradation can be specified, or indeed even before an intelligent estimate of the feasibility of designing such a system can be formulated.

In Chapter II, the system organization of a multiprocessing structure is discussed along with the particular characteristics of such a structure which facilitate reconfiguration and graceful degradation and the unique logical design problems attendant to the achievement of an ultrareliable multiprocessor. Maintenance policies, which reflect the computation and design constraints defined in Sec. I-A, are discussed and a flow description is given, which points out the system response to various error, input and interrupt conditions.

In Chapter III, we summarize an initial effort to perform a detailed logical design of the various functions associated with the selected computer organization. It appears that the reliability will be enhanced if some capability for repair is incorporated into the memory, processing, and control units. Concerning the processing units and possibly a portion of the control and memory units, the most attractive repair scheme is one for which the logical realization is a one-dimensional cascade of identical elements, whence the repair operation requires only the routing to a succeeding element in the cascade of the signals destined for a faulty element We call such a logical realization a byte-sliced realization since it is natural to assign a byte (containing at present an undetermined number of bits) of each of the registers, adders, decoders, etc. to each element or slice in the cascade. A design of a byte-sliced microprogram control unit is described, and a memory module, which embraces byte slicing in addition to such reliability enhancement techniques as data channel coding and access switch-failure detection is described. A novel digital realization is given, which relates to the voting-switchover scheme described in detail in Ref. 1.

Chapter IV is concerned with the design of commutation networks for the various data switching functions associated with a byte-sliced multiprocessor. In particular, we consider permutation networks that, for example, route data from a selected memory to a selected control unit, and order-preserving networks that route data between the working (i.e., failure-free) byte slices of memory and control units. An important feature incorporated in the designs is the capability to accommodate for hardware failures in the commutation networks without requiring the discarding of the entire memory and processing units or even the byte slices served by the network.

Chapter V is concerned with what seems to be an initial attempt to specify formal rules for the synthesis of programs in an ultrareliable computer. The description is in a general framework so as to embrace two important problems. The first of these is the synthesis of programs

which are mistake-free, wherein a mistake is assumed to be the result of human frailty, or of programs for which mistakes are readily detected and recovery to correct execution is possible. The second is the synthesis of programs that will facilitate the detection of hardware failures. We also consider the interrelationship between hardware and software, in particular, the ancillary hardware that will facilitate the specification of reliable software.

II PRINCIPLES OF MULTIPROCESSOR SYSTEM DESIGN

A. Introduction

In Sec. I-A a brief list was presented of the characteristics of advanced spaceborne computation and the constraints on a computer design that are a consequence of these characteristics. It is observed--although clearly the observation is not original--that the multiprocessor organization provides an excellent match to the severe spaceborne computer design constraints. The basis for this conclusion is that in a multiprocessor structure:

(1) The facility exists for simultaneous execution of several programs

(2) It is possible to satisfy a wide range of computation time requirements by assigning a varying number of processors to a given task

(3) It is possible to satisfy a wide range of accuracy* requirements for the same reason as in (2)

(4) Reconfiguration, to enhance reliability, is easily accommodated, at least at the processor level

(5) Accommodation to different urgency* levels is easily achieved by assigning a varying number of processors to a given task, although in this case each processor will operate with the same set of data†

(6) The data switching does not appear to be significantly more complex than would be expected for a single processor structure with extensive reconfiguration capability.

Reference 1 concluded that the key problems of system design in achieving a reliable reconfigurable computer are flexibility of structure, modularity, simplicity of diagnosis, and reliability of control.

---

* These terms were informally defined in Sec. I-A.

† It is implied that a set of processors are operating in the replicated mode, and the data outputs are in some way compared.

The first two problems, at least at the processor level, are clearly solved by the multiprocessor organization; the solution to the latter two problems must await the detailed study of maintenance policies and of the logical design of the processors.

Many descriptions of multiprocessor systems have appeared in the literature,[8,9,10] and several contemporary computer systems[11] rely upon multiprocessing. Most of these previous descriptions have been concerned with (1) gross estimates of system reliability, assuming for example, that diagnosis and switchover are always executed correctly; (2) scheduling analyses and simulations to facilitate the determination of system responses to various inputs; and (3) the specification of software that will enable the optimum utilization of the hardware. Our intention in this program is to study maintenance policies and logical designs that will maximize the system reliability,* and then to formulate a realistic estimate of system reliability, which will, hopefully, compare favorably with the previous gross estimates.

In this chapter we discuss a possible multiprocessing system organization, considering those policies which apply particularly to error control and the functional requirements of the various blocks.

B. Multiprocessor System Organization

The multiprocessor model with which we are concerned, is depicted in Fig. II-1. This system consists of a set of M high-speed working memories (WM); a set of N simple processor and control units (SP); a set of Q arithmetic logic units (ALU); a set of R back-up memories (BAM); an input-output device controller (I/O) for which we provide sets of spare registers, counters, buffers, and real-time clocks; two commutation networks (CN); a

* At present the "reliability of a multiprocessor" has not been formally defined, but we will temporarily assume on a qualitative basis that the reliability measure reflects the capability of the unit to carry out the set of mission computation tasks weighted according to priority, urgency, and accuracy.

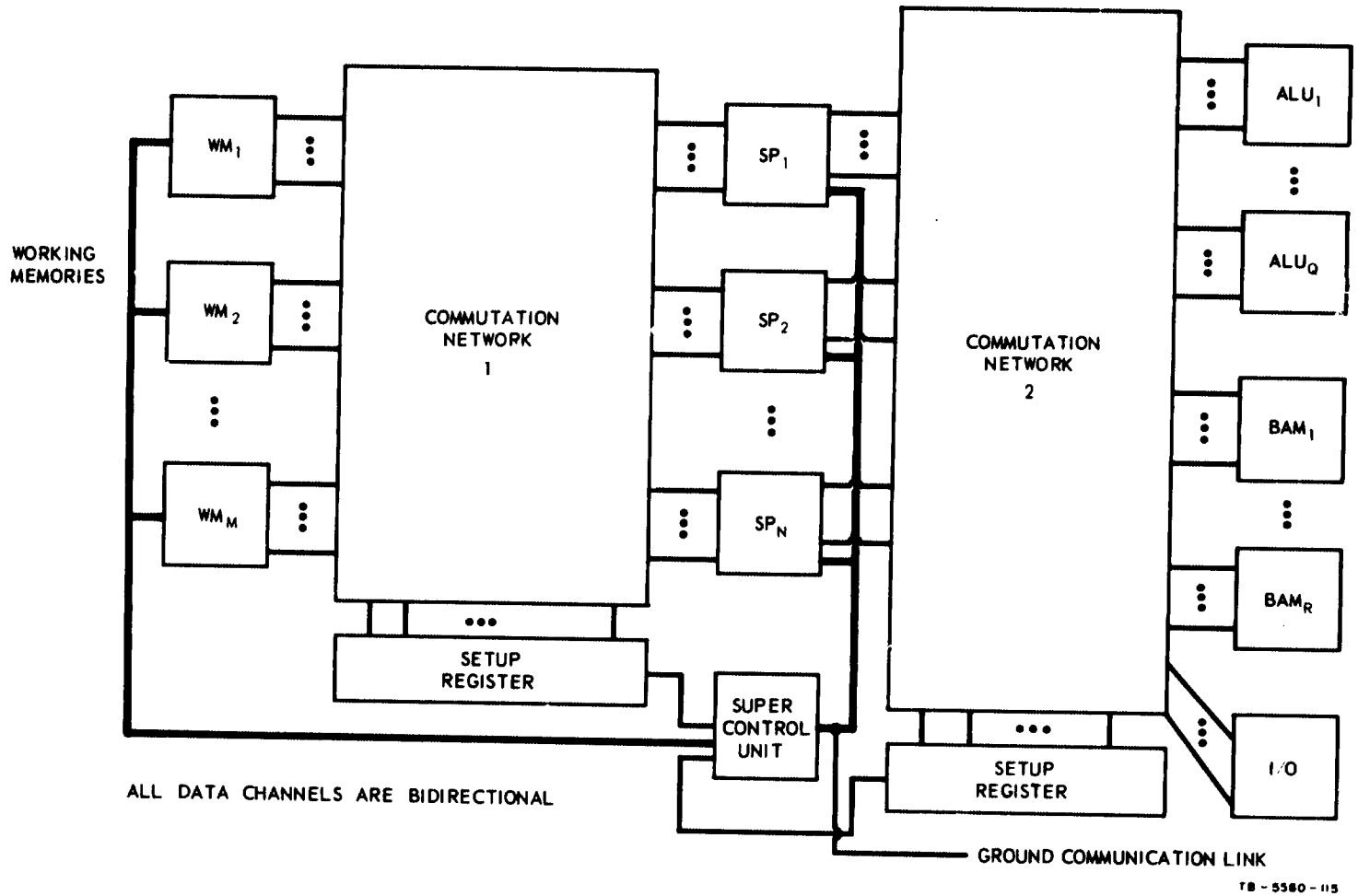


FIG. II-1 MULTIPROCESSOR COMPUTER SYSTEM BLOCK DESIGN

supervisory control unit (SCU); and two registers for setting up the commutation networks (it is convenient to view these registers as forming a component of the SCU).

In operation a given set of WM's, SP's, and ALU's will be in communication by means of links established in the two commutation networks. In addition, the commutation networks* perform the function of directing data to the failure-free byte slices, thus, effectively "repairing" the units.

It is envisioned that each SP unit will have the capability of executing comparatively simple decision and arithmetic algorithms, the capability of controlling program flow, and the capability of controlling processor allocation and scheduling.† An ALU will be used for the execution of complex algorithms requiring extensive processing hardware. The SCU will function as a referee in all error-control processes, and, essentially, represents the system hard-core, although it can be superceded by a command from the ground. The BAM's will store the task programs, diagnostic programs, and the setup programs for the commutation networks.

In addition to the inter-unit communication links provided by the commutation networks, a single data channel (probably serial), shown as bold-faced lines in Fig. II-1, is provided. This data channel links the SP and WM units with the supervisory control unit. It was noted by Alonso[10] that a complete multiprocessing system could be designed, containing only this single data-channel communication link, although in practice it is doubtful that this link would be serial. However, we have included the possibility of multiple-simultaneous communication between SP and WM blocks because of the additional flexibility thus provided, and because it seems appropriate, at this stage, to work with a general model.

---

* Detailed logical designs of commutation networks are presented in Chapter IV.

† It is implied here that each processor can function as an executive. This "floating-executive" technique, which is developed with greater detail in Sec. II-C, is also discussed in Ref. 12.

One important feature of the system, which is not shown explicitly in the figure, is that each defined block of the system will have at least one distinct power supply associated with it.* Furthermore, it is assumed that the power can be disconnected from a faulty block without resulting in the propagation of errors into connecting blocks due to excessive loading on the part of the disconnected unit.

C. Description of Error-Control Policies

In this section we will describe a possible set of error-control policies by indicating in flow-form the system response to various input and error conditions. It should be recognized that the system will not function exactly according to this description, but, by reference to this description, we are provided with a reasonably complete set of functional requirements for the various blocks.

In Table II-1 we present the flow description, and in Tables II-2, and II-3 respectively, we summarize the functional requirements of the supervisory control unit and the simple processor and control unit performing the role of the executive. At the conclusion of Table II-1, a set of comments are presented to clarify anomalies in the flow description and to point out some instances where there is some doubt concerning the optimality of the strategy selected.

---

* It was shown in Ref. 1 that the distribution of power supplies does not appreciably increase the weight or raw power requirements beyond that associated with the use of a single power supply.

Table II-1

FLOW DESCRIPTION OF MULTIPROCESSOR

1. SCU selects an SP (say SP_E), which is listed in the table of available SP's, to function as executive.
2. SCU instructs SP_E to access an available BAM.
   - 2.1 If accessing capability of SP_E has failed, SCU selects another executive, and SP_E is removed from the table of available SP's, and Step 2 is repeated.
3. SP_E selects an available WM to serve as the executive working memory, WM_E.
4. SP_E retrieves from a small store, possibly associated with the SCU, the setup data so that SP_E can access WM_E.
5. Setup data is transferred to the setup register of CN_1.
6. A WM and SP diagnostic program is directed from BAM through SP_E to WM_E.
   - 6.1 If the diagnostic program indicates SP_E cannot function as an executive, another SP is chosen and Step 2 is repeated.
   - 6.2 If diagnosis of WM_E fails and WM_E cannot be repaired, another executive memory is selected, and WM_E is removed from the available table.
   - 6.3 If WM_E is repairable, SP_E performs the task.
7. The executive program is transferred from BAM through SP_E to WM_E.
8. SP_E supervises diagnosis of the remaining set of SP's.
9. SP_E selects a set of WM's, SP's and ALU's to communicate with each other.
10. SP_E computes the setup data to effect this communication and transfers setup data to the two setup registers.
11. Each SP diagnoses its associated ALU and WM.
    - 11.1 If diagnosis fails and associated WM's and ALU's are not repairable, a different assignment is effected.

Table II-1 (Continued)

11.2 If the WM's and ALU's are repairable, the associated SP performs the task.

12. SP_E responds to an input device requesting service.

12.1 Assume urgency value of input is 1 (simplex):

12.1.1 SP_E selects SP_{S_1}, WM_{S_1}, ALU_{S_1} to handle program.

12.1.2 Pertinent program is transferred from BAM through SP_{S_1} to WM_{S_1}.

12.1.3 Input data is transferred to WM_{S_1} and also to WM_E.

12.1.3.1 Results of intermediate calculations might be stored in WM_E (as well as in WM_{S_1}) to provide a convenient roll-back point in the event of a failure.

12.1.4 Assume SP_{S_1} detects a computational error:

12.1.4.1 The computation is repeated, using the data and results stored in WM_E, to determine if failure is transient or permanent.

12.1.4.2 If error continues, the problem, data, and intermediate results are transferred to another set of SP, WM and ALU units for continuation of computation.

12.1.4.2.1 SP_{S_1} is diagnosed and either retained or discarded.

12.1.4.2.2 WM_{S_1} and ALU_{S_1} are either diagnosed by SP_{S_1} (if it does not contain the failure) or retained for other tasks.

12.1.4.3 If computation error continues with the replacement set of SP, WM, and ALU units, the error resides in the program or input data.

Table II-1 (Continued)

12.2 Assume urgency value of input is 2 (duplex):

12.2.1 SP_E selects the set of units (SP_{D_1}, WM_{D_1}, ALU_{D_1}) and (SP_{D_2}, WN_{D_2}, WM_{D_2}, ALU_{D_2}) to handle program.

12.2.2 Pertinent program and input data are transferred from BAM through SP_{D_1}, SP_{D_2} to WM_{D_1}, WM_{D_2}.

12.2.3 After each intermediate computation is completed, a change-commutation network command is directed to the SP_E.

12.2.4 The assignment of CN_1 is altered so that SP_{D_1} and SP_{D_2} exchange working memories.

12.2.5 The results computed by the two sets of units are compared.

12.2.5.1 The intermediate computation is repeated, commencing at the latest point in the program where the computations were in agreement.

12.2.5.2 If the discrepancy ceases, then the failure was transient.

12.2.5.3 If the discrepancy continues, then the failure is immediately attributed to one of the sets of units (SP_{D_1}, WM_{D_1}, ALU_{D_1}), (SP_{D_2}, WM_{D_2}, ALU_{D_2}).

12.2.5.4 The faulty set is disconnected, and CN_2 is altered by SP_E so that an available set of blocks are connected.

12.2.5.5 The faulty units are then diagnosed.

12.3 Assume urgency value of input is 3 (triplicated):

12.3.1 SP_E selects three sets of units (SP_{T_1}, WM_{T_1}, ALU_{T_1}) (SP_{T_2}, WM_{T_2}, ALU_{T_2}) and (SP_{T_3}, WM_{T_3}, ALU_{T_3}) to handle program.

12.3.2 Pertinent program and input data are transferred to the three working memories.

Table II-1 (Continued)

12.3.3 After each intermediate computation is completed, the results obtained by each of the three sets of units are compared.

12.3.4 If a unit's results disagree with those computed by the remaining two units, the dissenting set is disconnected and replaced by another set.

Comments

1. It is assumed throughout that the decisions of the executive are continuously checked by the SCU. For example, an executive that consistently requests the diagnosis of the other SP's would be disconnected from service and subsequently diagnosed.
2. The status of the SCU can be periodically monitored by, for example, a ground station, whence the SCU can be disconnected from service if it appears to be faulty. The ground station can then assume the "veto" power previously assigned to the SCU.
3. It has not been assumed for a general system that only a single unit can fail during the period between the execution of diagnostic routines, although a system which contained only two SP units would probably be disabled by the occurrence of simultaneous faults in each SP.
4. In the description, it is implied that all units are diagnosed immediately prior to insertion in the system. Although this policy would make a single-failure assumption seem more tenable, it is not strictly required since the system can accommodate a policy wherein diagnosis is deferred until a failure is detected.
5. In Step 9 it is assumed that a sufficient quantity of WM's, SP's and ALU's are available. If this is not the case, then several tasks might be executed with the same equipment in a

Table II-1 (Concluded)

multiprogrammed mode, or an SP unit might execute a program without recourse to an ALU. In addition, in order to satisfy stringent accuracy requirements, several ALU's might be assigned to function with a single SP in the execution of a program.

6. We have assumed that each task could be assigned one of three urgency values, corresponding to simplex, duplex and triplicated modes of operation. The simplex mode, wherein a processor detects its own faults, would be attractive for tasks that could be interrupted and for which the data and program code permit convenient roll-back to a known error-free state. The duplex mode, wherein two sets of units function simultaneously, would be used for interruptable tasks that require immediate error detection, but for which convenient roll-back is not possible, and for which an accurate record of all calculations must be available. The triplicated mode, wherein three (or possibly more) sets of units function simultaneously, would be used for the most critical mission tasks.
7. The reliability status of the various SP, WM, and ALU blocks is stored in a table of available equipment, which can be conveniently considered to form a portion of the SCU. A failure of this table could be circumvented by assuming that all units are faulty and whence each unit is diagnosed prior to insertion in the system.

Table II-2

FUNCTIONAL REQUIREMENTS OF SUPERVISORY CONTROL UNIT

1. Store a table of available equipment.
2. Select an SP to function as the executive, and direct the chosen SP to access a diagnostic program.
3. Control the diagnosis of an SP that will function as the executive.
4. Monitor all directives of the executive, but the SCU can only "veto" the commands of the executive.
5. Respond to earth commands to disconnect itself from service.

Table II-3

FUNCTIONAL REQUIREMENTS OF EXECUTIVE

1. Compute the set up data for the commutation networks.
2. Respond to all input and interrupt commands.
3. Select sets of SP's, WM's and ALU's for various tasks.
4. Organize the diagnosis and repair of units to be inserted in system.
5. Respond to the error and program status conditions of the other SP units.

D. Logical Design, Strategy, and Software Problems Associated with Multiprocessor-System Design

As indicated in Table II-1, many detailed problems of design and analysis are critical to the functioning of the multiprocessor. Among these are the following:

(1) A study of the simultaneous data transfer capacity required for the commutation networks. Of concern here are the trade-offs between rate of program execution and the complexity of both the commutation network and the setup algorithm.

(2) The design of commutation networks that offer economy of design, ease of setup, ease of diagnosis, and a tolerance to failures in the sense that a failure in a portion of the commutation network should disable a minimum amount of processor and memory capability. (See Chapter IV)

(3) The design of memory and arithmetic logic modules that offer combinations of fault masking and ease of reconfiguration. A discussion of such ALU's is presented in Ref. 1, and the techniques of microprogram control for such units is discussed in Sec. III-C of this report. A description of the organization of a reliable memory module is given in Sec. III-B of this report.

(4) The design of simple processor and control modules. It is envisioned that these modules would be of minimal complexity and would possess the capability of either controlling the flow of program data between a WM and an ALU, or perform the role of both controller and processor upon the occurrence of an insufficient quantity of available ALU units. A convenient framework for such a module is contained in a paper by Frankel,[13] which discusses the minimum complexity required for a digital computer. Some reliability enhancement might be incorporated within the module, either in the form of reconfiguration capability at the register and adder-byte level, or in the form of fault masking for irregularly structured control functions.

(5) The incorporation of protection for the back-up memories. If these memories are in the form of tapes, reasonably simple error-correction coding techniques can be applied for the protection of stored data.

(6) The synthesis of programs that are amenable to simplex type error detection, and the development of techniques which permit the detection of, and the recovery from, errors introduced by programming mistakes. (See Chapter V.)

(7) A reliability analysis of the overall multiprocessor system and an evaluation of the assumed maintenance policies in order to uncover a possible "optimum" set of strategies.

III TECHNIQUES OF LOGICAL DESIGN

A. Introduction

This portion of the report is concerned with schemes for the logical design of networks that realize major functions appropriate to a reconfigurable, reliable computer. The functions to which we have given the greatest attention thus far in the second phase of the study are memory, microprogram control, and adaptive voting. The important case of an arithmetic-logical processor was examined in the first phase. The approach taken was to organize a processor as an iterative array of byte-oriented modules, called byte-slices. This approach, which appears to be quite powerful, is applicable to a number of special networks (e.g., in microprogram control units). Further development of processor networks will be examined in the coming period.

B. The Organization of a Reliable Memory Module

1. Introduction

The working memory of a computer comprises a natural organizational unit for the application of error control. It is a vital unit, accounting for a large fraction of the equipment of a modern computer, and its simplicity permits a wide range of error-control approaches; hence, it is of interest to develop effective and flexible design techniques appropriate for memory systems.

In Appendix A of the Final Report--Phase I,[1] a number of techniques for error control in conventional working memories were reviewed. As a result, the following observations were made. First, it is generally impractical to apply fault masking at the circuit level to the circuits involved in memory selection and storage; hence, error-control techniques must be applied at the logical level. Secondly, there are several significant sources of error, including data channels, word selection (access) circuits, basic timing and power circuits, and the logical schemes most

effective for error control for each source are quite different from one another. Furthermore, both transient and permanent errors are significant.

In this section we will describe the organization of a memory module in which several kinds of error-control schemes, primarily logical, are incorporated. Only the basic schemes are presented. Further analysis is needed to determine the optimum allocation of redundancy in the several schemes.

We assume a model that is a magnetic core memory. Thus, we include a power-decoding-switch (access switch), for word selection, a passive, destructive-read, recording medium, a set of power drivers for recording, a set of sense amplifiers, and associated timing and power-supply circuits. The techniques to be described allow for other kinds of memory, such as nondestructive-read, or active-element storage, or even some cyclic memories, but we shall not attempt to provide full generality.

Experience has indicated the need to consider the following error types as very significant:

(1) Transient errors, primarily in the data-read channels, due to external noise or to internal, data-sensitive signal cross-coupling

(2) Permanent, single-element, independent component failures

(3) Permanent, multiple-element, nonindependent component failures (e.g., a cracked integrated circuit device)

For the present we ignore the timing and power-supply circuits, since their design problems do not seem to be unique to memory systems.

2. System Description

The memory system, which we consider, is described with reference to Fig. III-1. It is a memory system of W words, each containing B bytes of data, where a byte contains D binary digits. In this system, the following redundancy schemes are incorporated.

(1) The system is partitioned into B subsystems, each serving all W words. Each subsystem is an independent memory, containing its own access switch and drivers, store and data sense and drive amplifiers (in a complete design, separate subsystem power supplies and timing circuits

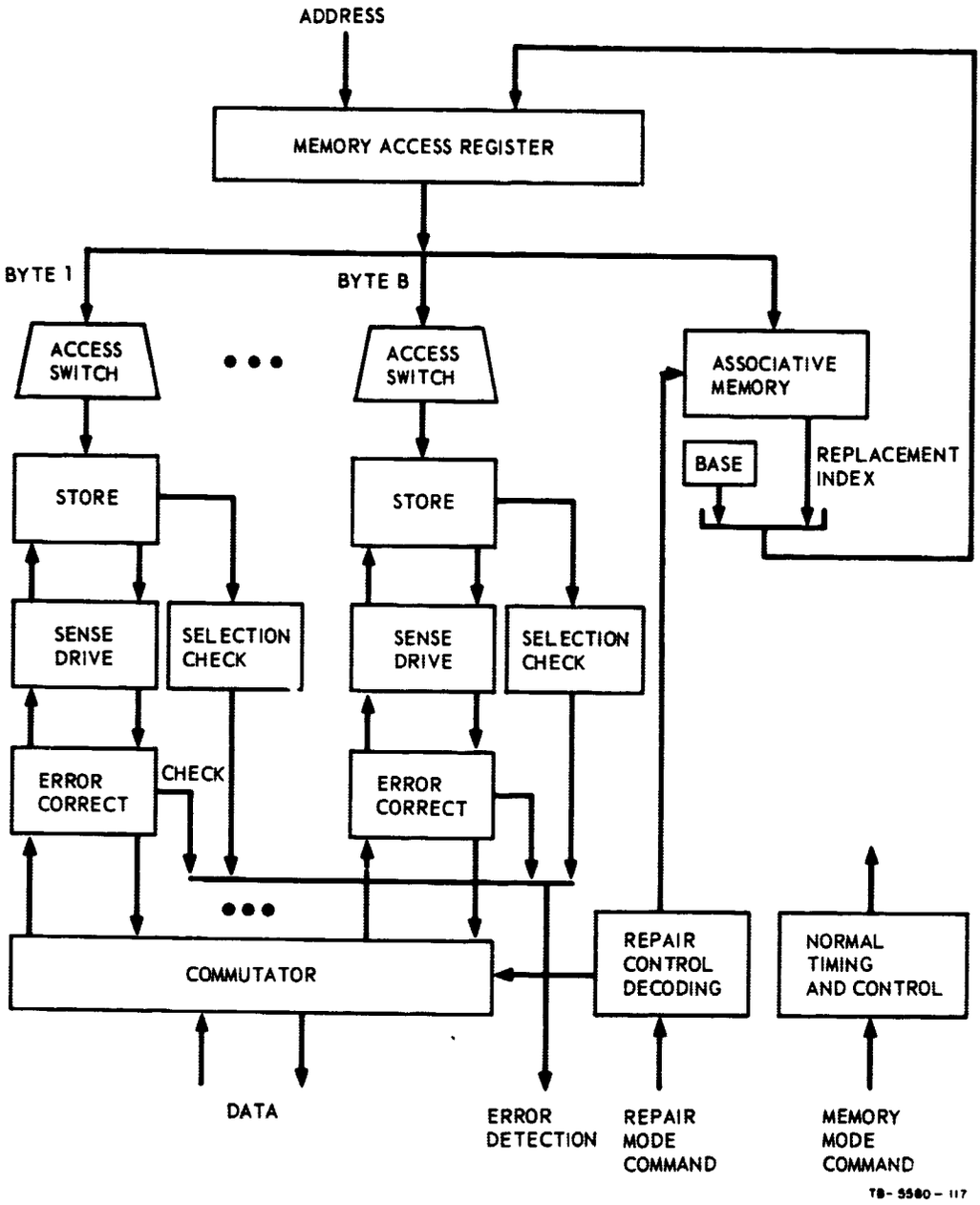


FIG. III-1 REDUNDANT MEMORY MODULE

would also be appropriate). B is greater than the number of bytes needed for a minimally acceptable word, B_N; hence, an order-preserving commutator network is provided to connect any subset of B_N out of the B bytes to the memory system interface.

(2) Incorporated in each store is a special circuit for determining the number of words selected at a given cycle, quantized to the levels 0, 1, and more than 1. Reliable circuits for this determination are well known. For example, one may provide one or two storage elements (cores) per word, operated so as to switch on a standard word selection excitation, with a common three-level sense amplifier. This circuit is very useful in determining whether or not the access circuits are faulty, since by far the most common modes of failures within access switches result in either no output or in multiple word selection.

This circuit is not absolutely essential since the cases of zero or multiple word selections could be inferred from analysis of the data channels. For example, if multiple parity-check redundancy is used (it is recommended in the next paragraph), the code could be designed to have a large error-detecting capability, and most multiple selections would be detected. However, the redundancy required for such error detection would be better used for error correction.

(3) The data of each subsystem are encoded with an error-correcting code. The most attractive codes for this purpose are those based on threshold decoding (see Sec. II, part B-2a of Final Report--Phase I[1]) because most single faults within the decoder network for such codes are masked, and the codes are reasonably efficient. Furthermore, as discussed in the Final Report--Phase I, double-error correcting codes are far superior to single-error correcting codes with respect to reliability-weight tradeoffs. The disadvantage of the former lies in the doubled cost (per data bit) of the decoding network, but with the new LSI technology, this is not a serious disadvantage.

An important consideration here is the problem of multiple transient errors due to environmental noise. The problem that arises is that if there are more than $t + 1$ errors in a word in a $2t + 1$ error-correcting system, the resulting pattern may appear (falsely) as a valid symbol, or as a symbol with t or fewer bits different from a valid symbol. As discussed in the Appendix,[1] this problem is reasonably well solved by using the error-detecting capability of the code; thus, the error-correction logic should provide an alarm when more than t bit errors are computed in a $2t + 1$ error-correcting code.

(4) An associative memory is employed to provide relocation for words that become unusable. The primary failure source for which this scheme is intended is the access switch and its drivers, although it is also useful for the case in which only a small number of storage locations suffer more bit damage than can be accommodated by the error-correcting codes on the data channels. In the scheme, a block of words, say the last 2^a words in a 2^c-word memory, are reserved for relocated data. An associative memory with capacity for 2^a entries is driven in parallel with the main memory subsystems. If one of its entries is excited, indicating that the external computer is addressing a word that has been relocated, a substitute location number, or alias, is emitted. For economy of storage, only a bits are needed for the alias since the high order digits may be provided by a constant (c - a)-bit number source; thus, the associative memory contains 2^a words, of c + a bits each. The high- and low-order digits are combined, and the resulting number replaces the original address in the memory access register.

3. Coordination of Error-Control Modes

Mode 1, partitioning with switch-over replacement, would be used when the error-correcting capability of all lower-level schemes is exceeded. It requires external diagnosis and control. Mode 2, detection of over- or under-selection, is important in preventing the data errors, due to access switch failures, being falsely corrected by the error-correcting logic in the data channels. Mode 3, error correction in the data channels, is the primary means of masking transient error. Mode 4, word-relocation, is important for accommodation of access-switch faults. It appears to be a valuable scheme, because, in commonly used switches, the average single fault results in the loss of only a small fraction of the outputs. The value of the scheme would be increased further, if the switch were designed so that the maximum fraction of outputs lost, due to any single switch fault, was limited.

A reliability analysis of the total system is needed in order to determine the optimum allocation of redundancy among the various error-control modes.

C. Design Techniques for a Modular, Microprogrammed Control Unit

1. Introduction

The use of a microprogram (μ-program) structure for the control unit of a reliable computer is attractive because it simplifies the task of modifying the behavior of the control unit in order to accommodate system failures. In addition, the inherent modularity of the major part of the structure potentially allows the use of highly efficient forms of error control within the control unit itself.

In this section we shall examine techniques for improving existing μ-program schemes in two ways that are significant to reliable computers; these are, increasing structural modularity, and increasing the effectiveness of reprogramming for system reconfiguration. The important problem of applying error control to the new schemes will be considered in subsequent work.

In a μ-program control unit, a control algorithm is represented by data called μ-instructions. These data are usually treated as words stored in an addressable memory. In practice, this memory may be realized as a matrix of logic elements (e.g., diodes), and frequently logic operations other than simple memory functions are employed within the matrix. In this study we assume that only memory functions are allowed. This assumption not only increases the generality of the results, but it permits the use of the main working memory of a computer as a back up for the μ-program data store.

The basic components of a μ-instruction are as follows:

(1) Specification of the active system data paths

(2) Specification of the operating modes of functional units

(3) Specification of the rules for selection of the next μ-instruction.

The third component is the source of most of the nonmodularity of structure. In the next section we discuss various kinds of next-instruction rules, and means for increasing modularity of implementation.

2. Schemes for Selection of the Next μ-Instruction

There are two aspects to the selection of the next μ-instruction. These two aspects, the composition of the address code and the generation of test functions for branching-type μ-instructions, will be discussed separately.

a. Composition of the Address Code

The address code may be composed, using data stored explicitly in the μ-instruction, or data stored in temporary data registers on counters, which may be considered to be implicit in the μ-instruction, or by some combination of explicit and implicit data. The explicit form has the advantage of speed, since no time is needed for loading special registers, and the disadvantage of storage cost. There is a wide variety of useful schemes, employing combinations of implicit or explicit references.

It is convenient to distinguish four types of μ-instructions, nonbranching, unconditional branching, two-way branching, and multi-way branching.

The explicit form is feasible and commonly used for the first three of the four types. For two-way branching, providing memory space in each μ-instruction for two explicit next-location fields may be very extravagant if only a small fraction of the μ-instructions are of this kind.

The implicit form is appropriate for the first type, nonbranching, μ-instructions; the natural rule is to take the new location as the previous location plus one. The second type, unconditional branching, μ-instructions require explicit data. In simple control units, the fraction of unconditional branching type μ-instructions may be low enough to justify placing the next-address data in a separate memory word so that a given word holds either external control information or next μ-instruction information.

For the third type, two way branching, μ-instructions, one of the next-address codes may be implicit (present address plus one). Another important scheme is to use two implicit addresses in a "skip" manner, as follows:

first branch: next address = present address plus 2

second branch: next address = present address plus 1.

In the second case, the next μ-instruction clearly must be an unconditional branch, the second type. The "present address" may also be explicit.

For the fourth type, multi-way branching, μ-instructions, the "skip" sequencing scheme may be generalized as follows:

next address = present address plus N,
where N is generated as a result of the test.

All but one of the next μ-instructions must, again, clearly be unconditional branches. Again, the "present address" may be explicit.

The foregoing descriptions illustrate the wide range of choices that are present in the design of the address-composition function. Inherently, all of them are consistent with a modular structure, since they require only the operations of register transfer, counting and addition. The particular choices require detailed examination of engineering trade-offs of speed and equipment cost within the context of a particular system.

b. Generation of Test Functions for Branching-Type μ-Instructions

Let $(x_1, x_2, \ldots, x_n)$ be the set of system status indicators (arithmetic overflow, input/output requests, program symbols, etc.)

There are several functions of these indicator variables that are of practical importance in the testing of system status. For the case of a binary test, i.e., for a two-way conditional-branching μ-instruction, the test function is a single variable, say f. This function, f, may be a general boolean function, but some functions are of special practical importance. These are $f = x_i$ and $f = x_{i_1} x_{i_2} \ldots x_{i_b}$, where x_i may be the true or complement form of a system status variable. Since f may be

tested to be true or false (and the x's may be complemented), the second form is equivalent to the form $f = x_{i_1} + x_{i_2} + \ldots + x_{i_b}$.

3. Schemes for Programmable Selections

In order to expand the power of the branch instruction, it is desirable to generate f under program control.[14] This also has the obvious advantage of enhancing reconfigurability for error control. This may be accomplished by providing a function-selection vector, say $\underline{T} = (t_1, t_2, \ldots, t_s)$, as a subfield in a μ-instruction word, and a logic network with inputs $\underline{T}$ and $\underline{X} = (x_1, \ldots, x_n)$, and output f.

A convenient network for this purpose may be built, using a decoder with $n \leq 2^s$ outputs and a simple AND/OR network as shown in Fig. III-2. An extension of this scheme, which provides for a programmable reassignment of selection code, is shown in Fig. III-3. A Σ stage acts as a +1 adder, such that output $T_{i+1} = T_i + 1$ (arithmetic sum) if $r_i = 1$, or $T_{i+1} = T_i$, if $r_i = 0$. An arithmetic overflow, y_i, is produced if $T_i = 2^s$. Furthermore, only one y will be 1. The cascade of U stages serves to select the x corresponding to the uniquely energized y, and to deliver it to the output f. The logic for a U stage is $u_{i+1} = u_i + y_i x_i$. The $\underline{r}$ variables serve to program the selection, since $r_i = 0$ causes stage i to be bypassed. This way of reassigning T codes to X variables permits reconfiguration without changing the T data in the μ-program memory. This may be advantageous, since it permits use of a read-only memory.

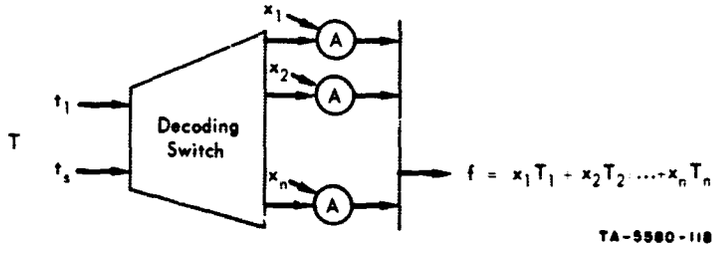


FIG. III-2 FIXED-STRUCTURE SELECTION NETWORK

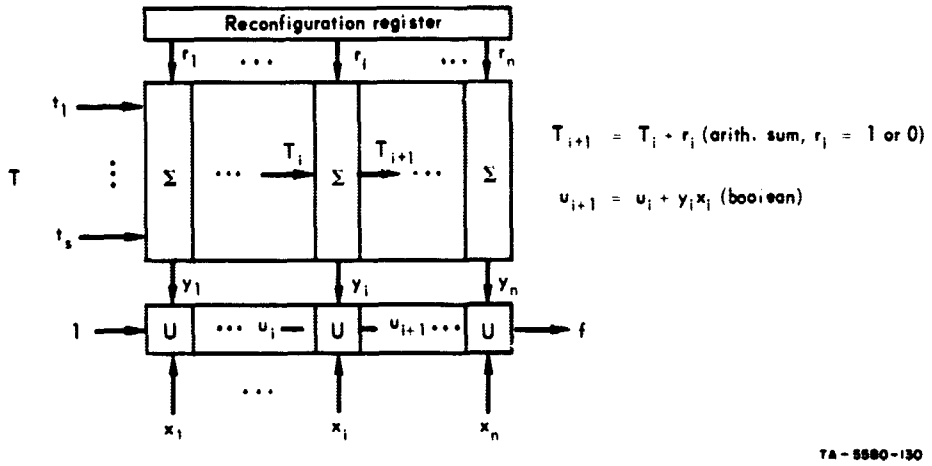


FIG. III-3 RECONFIGURABLE SELECTION NETWORK

4. Schemes for Programmable Generation of Boolean Implicants

A high degree of modularity and programmability in the generation of boolean implicants may be achieved by using a certain functional unit that appears to have great utility in modular computers. As shown in Fig. III-4, this unit consists of a single binary parallel adder and a scratchpad memory. (Such memories will be widely available in LSI form, and, in fact, the functional unit itself is a good candidate for LSI.)

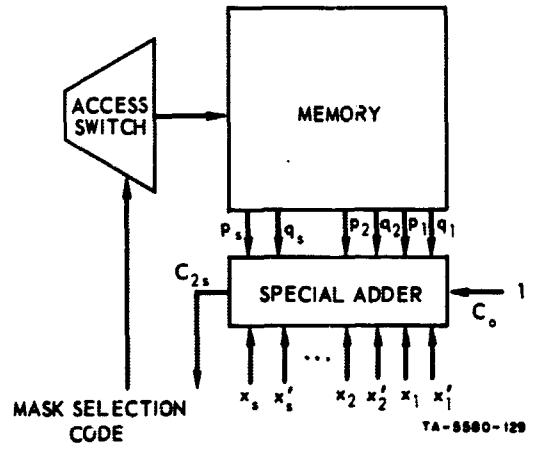


FIG. III-4 PROGRAMMABLE BOOLEAN-IMPLICANT FUNCTION NETWORK

In this application we employ the following carry function for each state:

$$c_i = (a_i + b_i)c_{i-1} \quad ,$$

instead of the usual arithmetic function,

$$c_i^* = (a_i + b_i)c_{i-1}^* + a_i b_i \quad .$$

If a standard arithmetic unit is used, the term $a_i b_i$ must be suppressed under separate control.

Let the set of system status variables be $X = (x_s, x_{s-1}, \ldots, x_2, x_1)$. We wish to evaluate boolean implicants, using any specified subset of the input variables. For example, we may wish to determine if $x_5 x_3' x_1 = 1$. We apply both the variables and their complements to one side of the adder, e.g., in the order $(x_s, x_s', x_{s-1}, x_s', \ldots x_2, x_2', x_1, x_1')$, with x_1' at the least significant binary input, and we apply to the other side of the adder a "mask" vector, obtained from the memory in response to an externally-specified address code.

Let the elements of the stored mask vectors be $M = (p_s, q_s, p_{s-1}, \ldots p_2, q_2, p_1, q_1)$, thus elements p_i, q_i correspond to the inputs x_i, x_i', respectively. If x_i appears in the implicant in true form, set $(p_i q_i)$ to (01); if it appears in complement form set $(p_i q_i)$ to (10), and otherwise, set $(p_i q_i)$ to (11).

To carry function at the $2i^{th}$ stage is

$$c_{2i} = (p_i + x_i)(q_i + x_i')c_{2i-2} \quad ,$$

thus, depending on the values of p_i and q_i, we have

$$c_{2i} = x_i c_{2i-2} \quad , \quad x_i' c_{2i-2} \quad , \quad \text{or } 1 \cdot c_{2i-2} \quad .$$

If the addition is carried out with c_0 set to 1, the final carry c_{2s} will be 1 if and only if the implicant is true.

As an example, let s = 5, and let the desired implicant be $x_5x_3'x_2$. The appropriate mask vector is M = (01, 11, 10, 01, 11). The boolean vector sum $\underline{M} + \underline{X}$ is (1, 1, ..., 1) (necessary and sufficient for $c_{2s} = 1$) if and only if $x_5x_3'x_2 = 1$.

A similar operation may be used to obtain the boolean sum of an arbitrary set of variables, each complemented or not. For example, in order to determine if $(x_4 + x_3 + x_1') = 1$, a test is made to determine if $x_4'x_3'x_1 = 0$. This is done by creating a mask for $x_4'x_3'x_1$ as described previously, and observing if the final carry c_{2s} is $\underline{0}$ (rather than 1).

More complex expressions may be realized using a single flip-flop to store the results of a succession of implicants. If the flip-flop is initially set, and if a zero carry resets the flip-flop, a given sum of products will be true if and only if the flip-flop remains set following the last product test.

5. Hierarchy Schemes

The simplest structure for a μ-program unit provides for the direct application of the binary elements of the μ-instruction word to the external control lines. This structure is used in practice, but there are some advantages to be derived from the use of a hierarchical structure.

One form, proposed by Grasselli,[15] is analogous to the so-called interpretive programming systems. In this form, illustrated in Fig. III-5, the number of different μ-instructions is limited, and a compact code is established for their representation (each word is called a μ-order). A μ-program is recorded in a first memory as a string of μ-orders. A "dictionary" of μ-instructions is held in a separate memory, and, in operation, a selected μ-order serves to address that memory. The merits of this scheme are several. First, most of the data storage, i.e., the dictionary, may be held in a fixed memory, while still preserving some

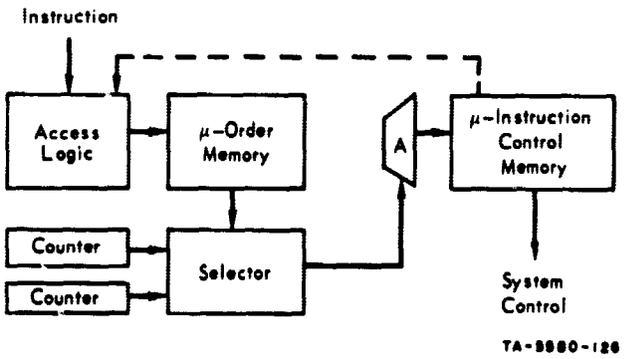


FIG. III-5 TWO-LEVEL MICROPROGRAM SCHEME (after Graselli)

flexibility in the μ-program sequence. Secondly, since the μ-orders are in a compact code, several may be packed into a single word of a relatively slow memory (e.g., the main working memory) without unduly slowing the control sequences.

Another form to which the authors know of no prior reference of special value in error control is to provide some kind of modifiable translation on the components of the μ-instruction word. The purpose of this translation would be to change the assignment of activation signals to the external functional units, thus effectively reconfiguring the system. This translation would be effected in a special network (a useful form for such a network would be an associative memory). This scheme also has the merit of allowing the main memory to use a fixed data store. Another advantage is that for high degrees of redundancy, the capacity of the μ-instruction store is minimized, since only the nonredundant control codes need be recorded. Figure III-6 illustrates the application of this method.

Both hierarchy schemes have attractive features, but further investigation is needed to determine the costs in speed and amount of logic needed for their implementation.

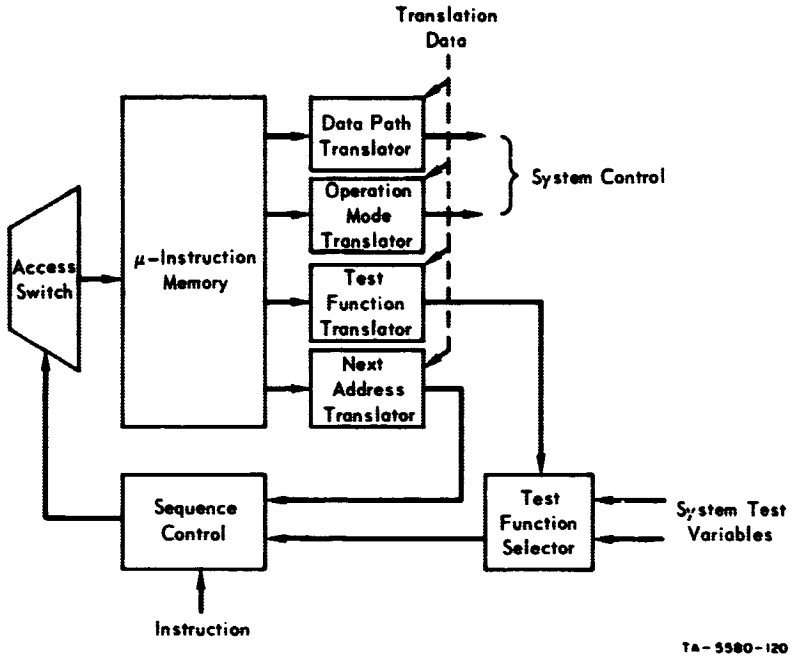


FIG. III-6 FIXED PROGRAM-VARIABLE TRANSLATION MICROPROGRAM SCHEME

6. Conclusions

Various alternatives in the design of μ-program type control units have been reviewed. A number of schemes have been developed for realizing major functions in modular and programmable forms and for permitting a substandard use of fixed-data type memory stores. However, the merits and costs of the various schemes need to be evaluated. It is also necessary to investigate appropriate means for applying logical redundancy for error control within the control unit.

D. An Improved Realization for Switched-Adaptive Voting

1. Introduction

In the von Neumann voting scheme, up to m errors in the outputs of $2m + 1$ nominally identical binary channels may be corrected by obtaining the system output as the majority function of the channel outputs. Pierce demonstrated the value of combining the outputs by a function that weights

the contribution of each channel according to its recent error rate. A continuous weighting function is discussed on page 371 of the Phase I, Final Report. As a special case, it may be seen that it is beneficial to completely disconnect a permanently-failed channel.

Several logical realizations of the latter scheme are also described in Sec. IIA, 2b of the Final Report--Phase I. The simplest structure was obtained by use of a linear-input logic element in which the 0 and 1 states of the inputs were encoded as +1 and -1 signals with the output 1 if the sum of the inputs is at least +1. Since linear input elements are difficult to realize in microelectronic technology, a realization was developed, using only binary-valued logic elements. This scheme required rather costly logic networks that effectively counted the number of active channels and also the number of channels displaying a 1 value. These counts were needed because the scheme called for replacing the output of a disconnected channel by 0; hence, the number of inputs (to the combining network) that constituted a majority depended upon the number of active channels.

2. Description of a New Scheme

The following approach permits the use of a fixed majority network as the entire combining element. Let only one channel be disconnected at a time, then, as successive channels fail, replace their outputs by constant 0 or 1 signals, alternately.

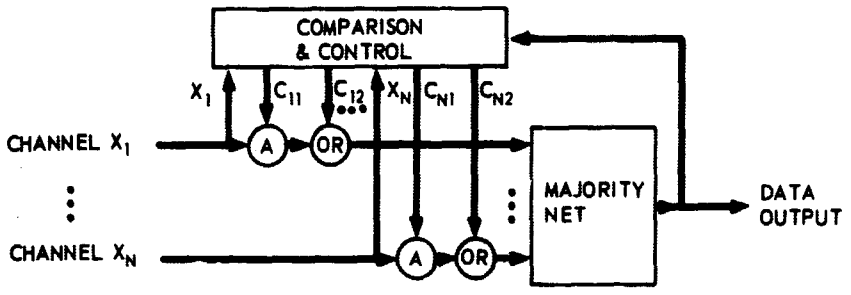
It is convenient to represent the inputs to the majority network as C_0, C_1, and x, representing, respectively, the constants, 0 and 1, and the free variable x. As an example, for the case $2n + 1 = 7$, the following useful sets of inputs appear at the input to the majority network: $(7x)$, $(6x, 1C_0)$, $(5x, 1C_0, 1C_1)$, $(4x, 2C_0, 1C_1)$, $(3x, 2C_0, 2C_1)$, and and $(2x, 3C_0, 2C_1)$. Since the system output is 1 when four 1's are

$(2x, 3C_0, 2C_1)$. Since the system output is $\underline{1}$ when four $\underline{1}$'s are input, the corresponding minimum ratios of free $\underline{1}$ inputs to total free inputs required for system output 1, are 4/7, 4/6, 3/5, 3/4, 2/3, and 2/2 respectively.

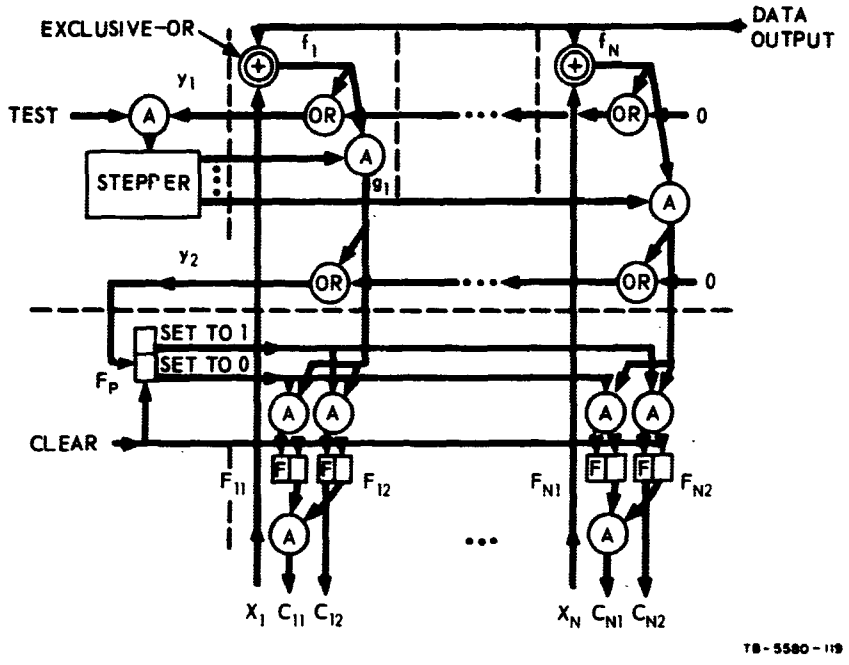
The overall structure of a realization of this approach is given in Fig. III-7a. Each of N channels, $x_1, \ldots, x_n$, is presented to a Majority Net via an AND/OR cascade, whose other inputs are obtained from a control unit. Thus, for example, the first input to the Majority Net is the function $(x_1)(C_{11}) + (C_{12})$. By proper choice of the control signals, C_{11} and C_{12}, the input may be set to the values x_1, 0, or 1. The Comparison and Control Unit determines when a channel is to be disconnected and also generates the appropriate control signals. Inputs to the unit are obtained from the channels and from the system data output.

A logical realization of the Comparison and Control Unit is illustrated in Fig. III-7b. The subnetworks corresponding to channels are arranged vertically, between dashed lines. The control outputs for a given channel are obtained from two flip-flops; thus, for channel 1, $C_{11} = \overline{F}_{11}\overline{F}_{12}$, and $C_{12} = F_{12}$. The Majority Net inputs x_1, $\underline{0}$ and $\underline{1}$ are produced by control outputs $C_{11}\overline{C}_{12} = 1$, $\overline{C}_{11}\overline{C}_{12} = 1$, and $C_{12} = 1$, respectively. These outputs, in turn, correspond to the flip-flop states $\overline{F}_{11}\overline{F}_{12} = 1$, $\overline{F}_{11}\overline{F}_{12} = 1$, and $F_{12} = 1$, respectively. Initially, all flip-flops are cleared to the $\overline{F}$ state, thus, setting all Majority Net inputs to the x state. In order to set the channel 1 Majority Net input to the $\underline{0}$ or the $\underline{1}$ state, either F_{11} or F_{12} is set to the F state, depending on the parity of the number of previously disabled states. This parity is recorded by a trigger flip-flop, F_p, the state of which is sampled by the signal g_1.

Signal g_1, which is derived from signal f_1 is gated by an output of the STEPPER unit. Signal f_1 is true whenever the channel data, x_1, and the system data output differ, the difference being taken as indicative of a channel error. The STEPPER unit serves to scan the channels serially; this is done to avoid ambiguity in the determination of the parity of active channels, in the case that several channels may fail at the same moment. The STEPPER is assumed to return to a rest state autonomously.



(a) OVERALL SCHEME



(b) DETAILS OF COMPARISON AND CONTROL UNIT

FIG. III-7 SWITCHED-ADAPTIVE VOTING

The presence of a difference in any channel is indicated by signal $y_1 = f_1 + f_2 + \ldots + f_N$, which, when true, starts the STEPPER, if permitted by the external control signal TEST. For each channel, if a g signal is produced, a $y_2 = g_1 + g_2 + \ldots + g_N$ signal is produced, which triggers the parity flip-flop, F_p. The information in F_p is also available in the set of F_{i1} status flip-flops.

In a system composed of a number of such voting stages, various portions of the control unit may be centralized. For example, a single

STEPPER unit could serve all stages, and in the extreme, all the control logic could be programmed, with the exception of the status flip-flops.

The minimum cost of the stage, assuming that, of the control unit, only the status flip-flops and their input gates are realized in hardware, is 2N flip-flops, 4N gates, and one majority network. The cost of the latter is about eight three-input gates for N = 5, and about eighteen three-input gates for N = 7. The total number is thus 10 flip-flops and 28 gates for N = 5, or 14 flip-flops and 46 gates for N-7. For comparison (also excluding the comparison logic), the scheme described in the Phase 1 report (pp. 74 and 85) requires about 5 flip-flops and 50 gates for N = 5, and 7 flip-flops and 95 gates for N = 7.

Assuming that a flip-flop is equivalent to four gates, the equivalent gate costs for five- and seven-input stages are 68 and 102, for the new scheme, and 90 and 123, respectively, for the old scheme. The availability of a three-state storage element would allow further economies.

IV PRINCIPLES OF COMMUTATION NETWORK DESIGN

A. Introduction

1. Commutation Requirements

In Chapter III, a multiprocessor system was proposed for which a set of simultaneous one-to-one connections could be established between working memory modules (WM) and simple processor and control modules (SP), and also between SP modules and arithmetic logic units (ALU) or back-up memories (BAM) or input-output devices (I/O). Also discussed was the possibility of "repairing" an SP, WM or ALU module wherein the module is realized in a byte-sliced manner. In such a case, the repair operation requires the routing of the data, previously destined for a faulty slice, to a succeeding slice. The networks that perform the data switching, which is inherent in the operations of module assignment and repair, are called commutation networks.

We have classified two types of data commutation for the module assignment, namely (1) complete permutation--complete utilization, (2) complete permutation--incomplete utilization, and three types of commutation for the repair operation, namely (3) incomplete permutation--order preserving, (4) incomplete permutation--nonorder preserving, and (5) "shorting." These five commutation functions are described schematically in Fig. IV-1, where the specific applications of each function are also listed.

The assignments associated with the complete permutation--complete utilization [CPCU(N)] function is probably obvious. The commutation network is to be capable of permuting, in an arbitrary manner, a set of N input data lines, emerging, for example, from a set of memories to a set of N output lines incident to, for example, a set of SP units. In the illustration, a data transfer path represents a parallel set of lines containing many bits (24-56). The assignments associated with the complete permutation--incomplete utilization [CPIU(N,m)] function

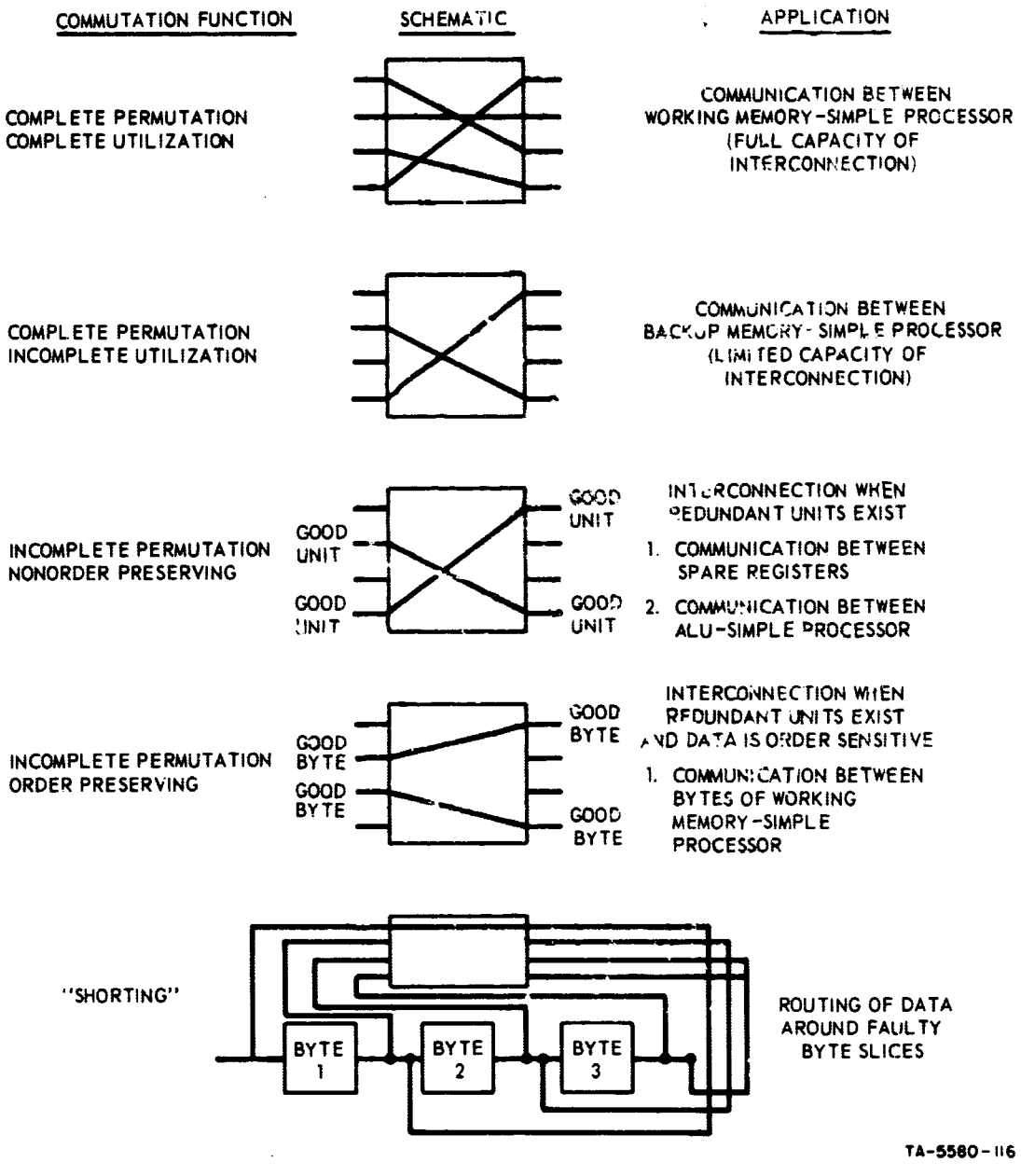


FIG. IV-1 CLASSIFICATION OF DATA COMMUTATION REQUIREMENTS

differ from those associated with the CPCU function in that for the former only a subset containing m inputs of the total of N inputs and outputs* needs to be interconnected at a given time. For the incomplete permutation--order preserving [IPOP(r,m)] function a subset containing m inputs of the r inputs, say for example associated with the working byte slices of a simple processor and control unit, is to be connected to a subset of the outputs, say associated with the working byte slices of an arithmetic logic unit, but with the restriction that spatial ordering of the input signals is to be preserved at the output. The preservation of order is clearly required since the data to be commutated is a binary number. The assignments associated with the incomplete permutation--nonorder-preserving [IPNOP(r,m)] function differ from those of the IPOP case in that, for the former, preservation of order is not a requirement. For the shorting function the outputs of a given byte slice are either to be connected to the succeeding stage (slice) or "shorted" around that succeeding slice.

For any commutation function it is desirable to synthesize networks for which the following are true:

(1) The design is economical

(2) The network setup is not difficult

(3) The data transfer is rapid

(4) Failures in the commutation network do not disable either the commutation network or the modules served by the network. This tolerance to CN failures should be achieved with minimal increase of network complexity.

(5) If the commutation network is "repairable," the diagnostic routines should be easy to specify and of minimal length.†

---

* We are assuming that for all commutation requirements there are an equal number of input and output terminals; this is clearly not a necessary restriction and it is only postulated for convenience in description.

† The "length" of a diagnostic routine is defined to be the maximum number of sequences of input data required to diagnose the network.

2. Prior Solutions to the Commutation-Network Design Problem

The obvious solution to the commutation-network design problem relies upon the use of a single-level crossbar switch, similar to the type commonly found in central telephone exchanges. Figure IV-2 represents, schematically, a crossbar switch serving a set of WM's and SP's. Here the crossbar, where each single-pole single-throw switch represents a crosspoint, fulfills the requirements of both the CPCU(N) and IPOP(r,m). Clearly a crossbar with N^2r^2 crosspoints would be sufficient, but it was shown[1] that actually $N^2(r^2 - m^2)$ crosspoints are sufficient. In any event, it is seen that $28 \cdot 10^4$ crosspoints are required to serve 25(=N) processors and memories, 32(=r) total bytes, and 24(=m) bytes are required for computation. Although a multiprocessor of this complexity might appear unreasonable, there is considerable motivation to seek more economical commutation network designs.

Goldberg[1,16] described a serial transfer network for the IPOP function which exhibited a complexity proportional to 2N, but, even with the

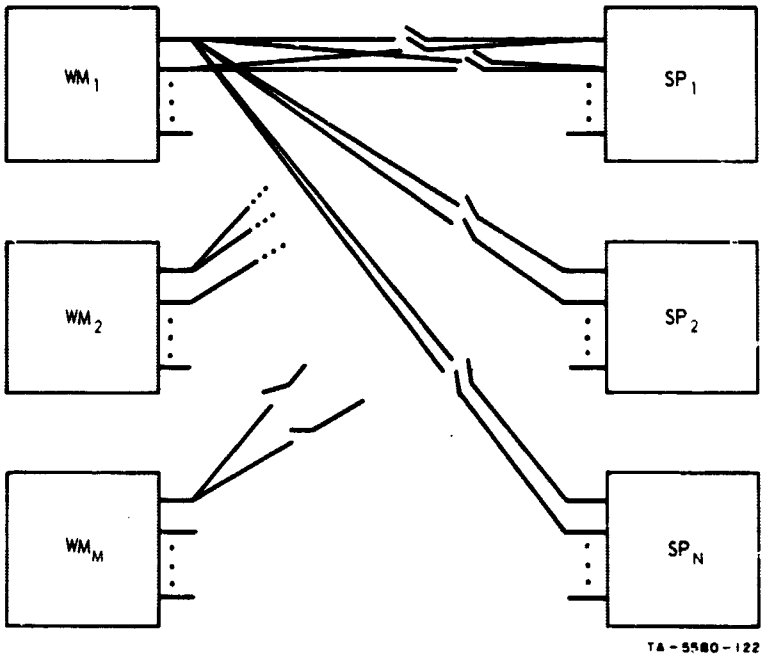


FIG. IV-2 "CROSSBAR" REALIZATION OF COMMUTATION FUNCTION

use of speed-independent logic, the data-transfer rate is probably not adequate for our application.

3. The Primitive Building Block of Commutation Networks

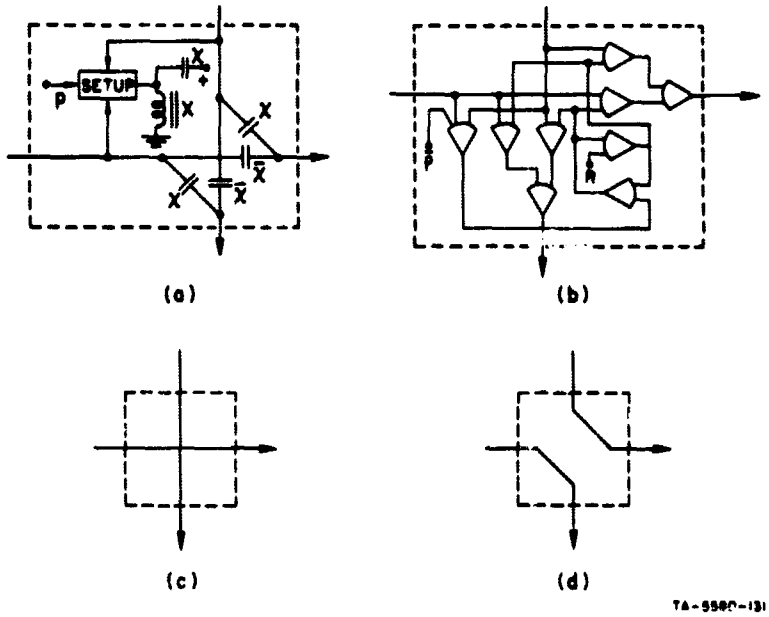
Most of the commutation networks described in following sections will be composed of interconnections of the "cell" shown in Fig. IV-3. The cell, which is very simple, behaves as a reversing switch, under the control of a single memory flip-flop.

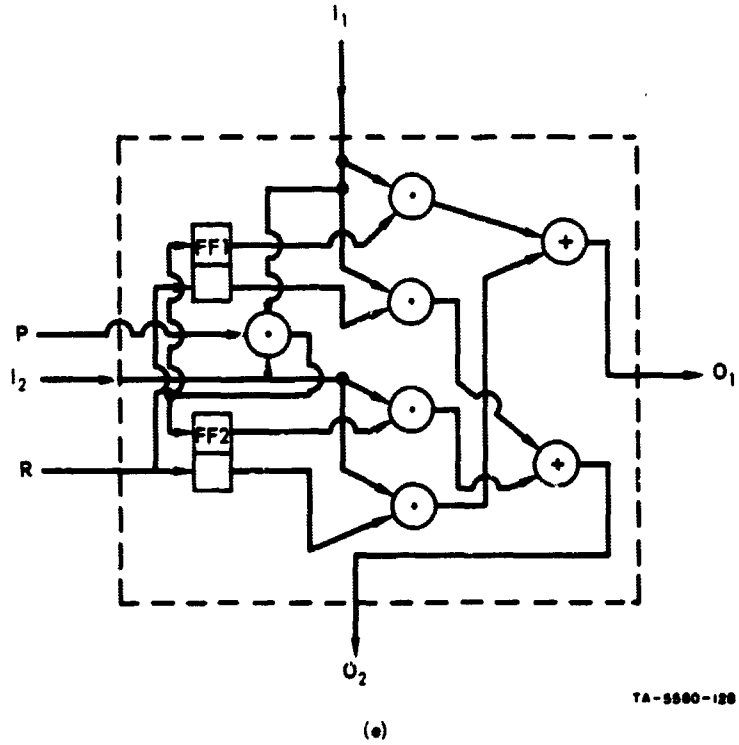
In essence, it is a double-pole, double-throw reversing switch, controlled by a storage element (e.g., a flip-flop). In addition some means is provided by which the storage element is set to the desired state. Figure IV-3(a) shows a relay-contact version, analogous to circuits in the MOS technology, and Fig. IV-3(b) represents a NOR gate-realization of the cell in question. The two modes consist of a crossing Fig. IV-3(c) and a bending Fig. IV-3(d) of the pair of input leads to the pair of output leads. Figure IV-3(e) represents a redundant flip-flop version of the cell, for which any single component failure will result in one of two possible failure conditions. In the first failure condition, which we will call the "stuck function" condition, the cell can realize only one of the two possible modes, i.e., the bend or the cross. In the second such condition, which we shall call the "bad-output" condition, one output lead contains a faulty signal. Table IV-1 summarizes the failure conditions resulting from various component failures of the cell of Fig. IV-3(e).

Table IV-1

FAILURE CONDITIONS FOR BASIC CELL

Component Fault	Failure Condition
Faulty OR gate	Bad-Output
Faulty 2-input AND gate	Bad-Output
Faulty 3-input AND gate	Stuck-Function
Flip-flop stuck in a mode	Stuck-Function
Same logic value on two outputs of a flip-flop	Bad-Output





(e)

FIG. IV-3 BASIC CELL FOR COMMUTATION NETWORKS

B. Commutation Networks for Complete Permutation--Complete Utilization

1. Nonredundant Networks*

It is easy to see that a CPCU(N) network must contain enough two-state cells† to specify all N! possible permutations; thus, $N_1(N)$, the number of cells in the network, is given by:

$$N_1(N) \geq \log_2 (N!)$$

or, asymptotically (from Stirling's formula),

$$N_1(N) \geq N \log_2 N - 1.443N + 0.5 \log_2 N \quad .$$

In Fig. IV-4(a) we depict a CPCU network. The subnetworks P_A and P_B are themselves complete interconnection networks, each with half of the number of inputs of the total network. This arrangement requires a number, $N_1(N)$, of cells which satisfies:

$$N_1(N) = N_1[(N/2)] + N_1[N - (N/2)] + N - 1 \quad .$$

Solution of this recursion for $N = 2^r$, [using $N_1(2) = 1$], gives

$$N_1(2^r) = 2^r(r - 1) + 1 \quad ,$$

or

$$N_1(N) = N \log_2(N) - N + 1 \quad .^{\S}$$

---

* The nonredundant networks for the case $N = 2^k$ described below were discovered under another contract, and are described in two forthcoming papers.[17,18] All of the other results were obtained under this present contract, as well as the generalization of the nonredundant network to cover the case where $N \neq 2^k$.

† For the CPCU applications of interest to us, the cell is actually an r-pole, double-throw, reversing switch, where r is the number of output lines contained in, for example, a WM. It is shown in Sec. IV-B-3 that it is convenient for reliability enhancement purposes to byte slice the CPCU network so that the number of "poles" each cell contains is considerably less than r.

§ It can be shown that the same value of $N_1(N)$ is obtained for arbitrary N.

A constructive proof that this network is capable of performing an arbitrary permutation is as follows. As previously, let the inputs and outputs be labelled X_1, X_2, ... X_N and Y_1, Y_2, ... Y_N, respectively, in order as shown in Fig. IV-4(a). We start by building a path backwards from Y_1 through P_A, and through whichever input cell is connected to the input that is supposed to be connected to Y_1. (The routing of signals within P_A and P_B will be determined in later steps.) A forward path is now formed from the other input $X_{Y_1 \pm 1}$ sharing the same input cell with X_{Y_1}, through this input cell, through P_B, and through whichever output cell is connected to the desired destination (Y_j, say) of this input line. In like manner, the mate to this new output is traced back through P_A to its associated input, and the mate of this new input is connected in a forward path through P_B to its intended output. This path-building process is continued, alternating use of P_A and P_B, until output Y_N is reached. If not all input-output connections have yet been completed, then start with any unconnected output line and continue the process, going through the cycle as many times as necessary, until all input and output cells are set in either the "bending" or "crossing" mode.

The entire procedures are now repeated separately for each of the interconnection arrays P_A and P_B, etc., until the entire array has been set up. The procedure will never be forced to stop because of the lack of connection, since P_A and P_B are each connected to every input and output cell--except Y_1 for P_B and Y_N for P_A, and these connections are avoided by starting as indicated. The array for $N = 8$ (with some of the connections straightened and the outputs renumbered) is shown in Fig. IV-4(b).

It is of interest to investigate efficient techniques for setting up the network cells to realize the necessary mode for a particular permutation. Consider, for example, the network of Fig. IV-4(b) redrawn in Fig. IV-4(c), and assume that we require the setting of the cells as shown. Referring to Fig. IV-3 we see that each cell is in the crossing mode upon resetting the flip-flop. The cell is set to the bending mode by the coincidence of logic-one values on the data inputs and the P input. Clearly, then, by applying a logic-one value to the P input of all cells in a given level of the network--in Fig. IV-4(c) a set of values, $X_5 = X_6 = P_1 = 1$,

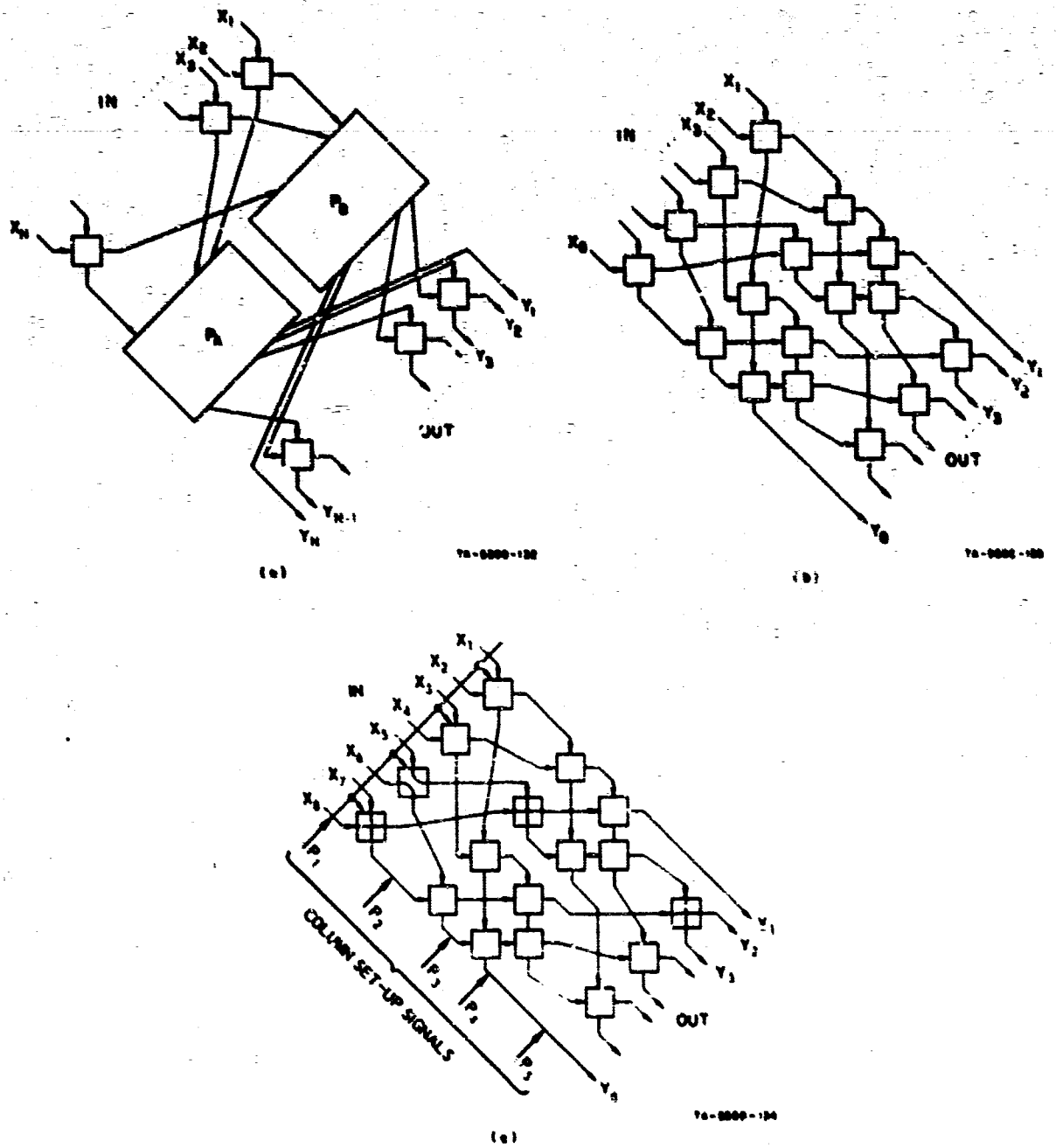


FIG. IV-4 NETWORK FOR COMPLETE PERMUTATION — COMPLETE UTILIZATION

will set the cell serving X_5 and X_6--and appropriately setting the N input signals, the network can be set up in a period proportional to the number of levels in the network.

2. Byte-Sliced Commutation Networks

In this section we are concerned with the behavior of the CPCU networks under cell fault conditions and a simple technique of accommodating to these faults.

It is clear that for the case wherein the basic cells are double-pole, double-throw, reversing switches, each bad-output cell failure results in an error only on a single output of the network. Similarly, each stuck-function failure results in an error on a maximum of two outputs. In the latter case, it is sometimes possible to accommodate to this failure type by appropriately setting the working cells of the network. This accommodation technique is discussed in detail in Sec. IV-B-3. Unfortunately, for the case of r-pole cells, a single failure could result in the inability of the CPCU network to realize several of the SM-SP assignments.

This state of affairs is significantly improved by byte slicing the CPCU network as shown in Fig. IV-5. In this case, the bytes (where each

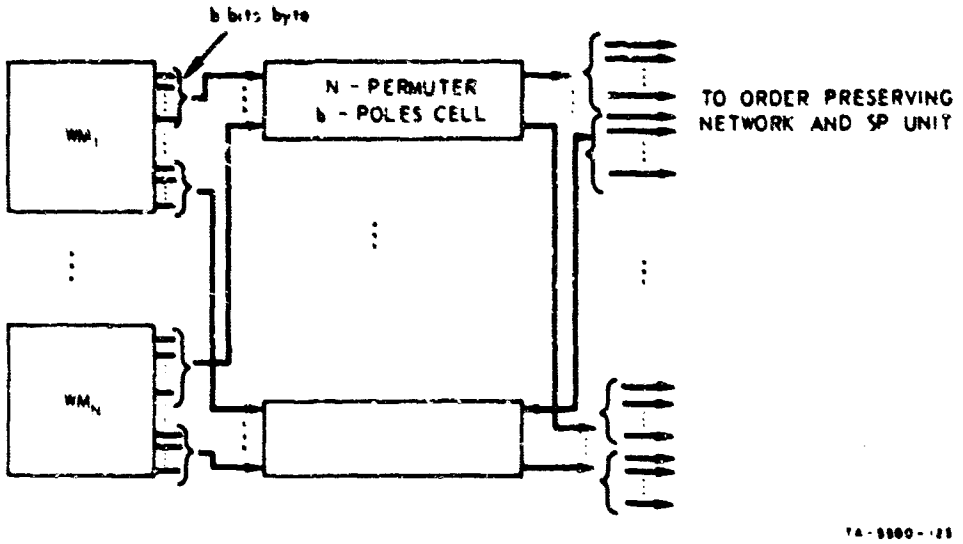


FIG. IV-5 BYTE-SLICED PERMUTATION NETWORK

byte is assumed to contain b-bits) of the WM's are permuted in separate networks. That is, the first CPCU has, as inputs, the first byte of each WM, the second CPCU has, as inputs, the second byte of each WM, etc. The outputs of the first CPCU are ultimately directed to first byte of each SP, etc. It is thus seen that for this byte-sliced realization, which, of course, requires the distribution of the cell memory flip-flops among each of the CPCU's, a cell failure disables a single byte. These commutation network byte failures can be accommodated for in the identical manner proposed for other byte failures, i.e., by the use of the incomplete permutation--order-preserving networks described in Sec. IV-D.

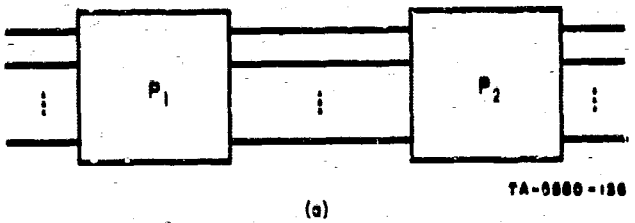
3. CPCU Networks Insensitive to Cell Failures

a. The Stuck-Function Fault

For the moment, consider only the case in which the network is fault-free or has precisely one bad switch (cell). Figure IV-6(a) illustrates a straightforward solution to the single error-correction problem. If P_1 and P_2 are both full permutation networks, then a fault occurring in one of them (such a fault being of the stuck-function type that does not disturb lead continuity) has no effect on the operations of the other network. Obviously, this is a rather wasteful approach since all of the remaining switches in the network containing the fault contribute nothing toward forming the desired permutation. Instead, let P_2 of Fig. IV-6(a) represent a permutation (CPCU) network, and P_1 be a network specifically designed to undo the damage caused by a fault in P_2. Then if a fault occurs in P_2, it can be repaired by P_1, while a fault in P_1 causes no trouble because P_2 is a full permuter. What is required of the network P_1? A single fault in P_2 causes a simple interchange of some particular pair of leads at some cell within the network. This can only become manifest as a spurious reversal of exactly two leads at the output. Of course, the possibility exists that the switch might fail in the correct position and no trouble would occur. In any event it would be sufficient that the network P_1 be capable of effecting the interchange of an arbitrary pair of input leads without changing the relative assignments of the other input leads.

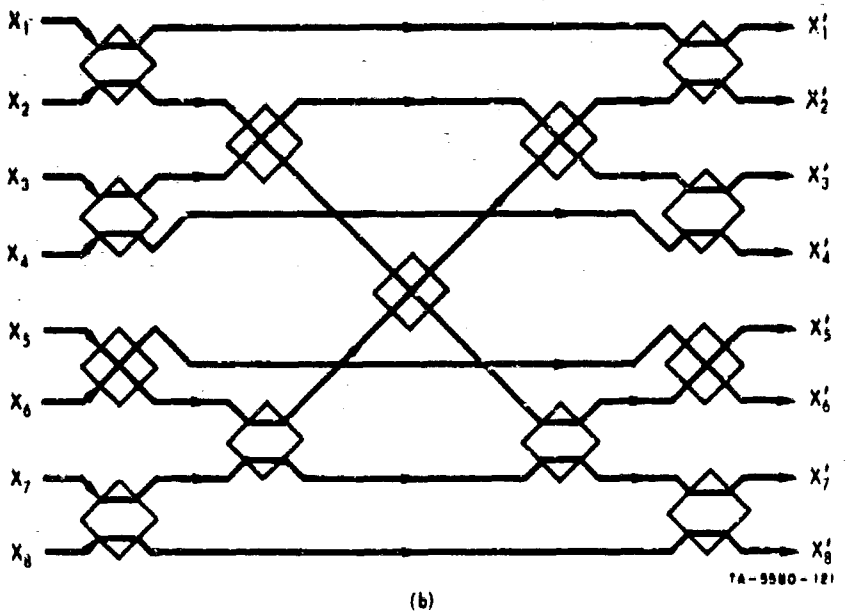
The double-tree (D-T) network of Fig. IV-6(b) can do this job. Any pair of input leads can be directed to some switch on the left of the center switch. At this switch (to the left of the center switch), the leads may be interchanged. Whatever switch settings are required to do this interchanging, we copy by reflection in the (imaginary) center-line to the right-hand part of the network with the exception of the switch that actually effects the interchange. The corresponding switch in the right-hand part of the network is set in the opposite state. The scheme is illustrated in Fig. IV-6(b) for the particular case of $N = 8$ in which we wish to interchange inputs X_2 and X_5. Here the switch settings to the right and left of the center line are identical and the center switch effects the desired interchange. Similar networks exist for all values of N, and are obtained by pruning the corresponding tree networks for the next largest power of two greater than N.

When one of these double-tree networks is placed in tandem with a full permutation network, we are able to correct the effect of one switch failure wherever it may occur in the composite total network formed by P_1 and P_2. If P_2 is one of the CPCU type commutation networks we have already considered, it will be found that the input peripheral switches of P_2 and the output peripheral switches of the D-T network match up into tandem pairs of individual switches. Note, for example, the pairing of leads at the input of the network of Fig. IV-4(b) and the similar pairing of output leads in Fig. IV-6(b). Whenever such a pairing occurs, we may omit one of the two switches if we have provided for the possible failure of one or the other of them. We may therefore omit the entire column of peripheral output switches from any of the D-T networks whenever we adjoin them to one of the CPCU networks. The single-error correcting capability is not affected by this pruning operation. Call the network that results from the removal of the output switches from the double-tree network the TDT(N) network (for truncated double-tree of N leads). Then single-error correction of any CPCU(N) network can be obtained by the tandem addition of one TDT(N) network. Furthermore, if the possible effect at the output of the CPCU(N) of the multiple failure of switches is considered, it turns out that the addition of p TDT(N) networks in tandem



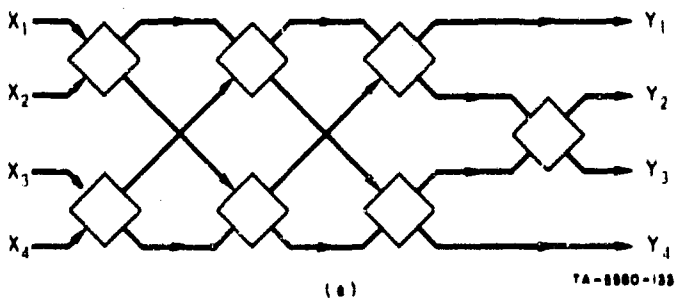
(a)

TWO PERMUTATION NETWORKS IN TANDEM



(b)

A "DOUBLE-TREE" NETWORK



(c)

MINIMAL REDUNDANT 4-PERMUTER —
SINGLE STUCK-FUNCTION CORRECTING

FIG. IV-6 PERMUTATION NETWORKS INSENSITIVE TO SINGLE "STUCK-FUNCTION" FAULT

to any CPCU(N) network will suffice to correct as many as p switch failures in the total network. The argument, which is much the same as the foregoing one, depends on the possibility of decomposing all such multiple failures into separate pairwise lead interchanges.

To estimate the cost of error protection according to the foregoing scheme, we note that the TDT(N) networks contain approximately (3/2)N switches. To correct p errors then takes about 3/2 (pN) switches. Thus, if N is very large, we can correct a "few" errors at a cost that is small compared with the total number of switches in the network ($\approx N \log_2 N$). On the other hand, correction of multiple errors with TDT(N) networks does <u>not</u> furnish a recipe for creating arbitrarily reliable networks (in the Shannon sense) while still meeting the asymptotic cell count, i.e., $\approx N \log_2 N$. We have not yet discovered how to resolve this problem although a solution seems possible. If anything can be concluded from results obtained for small values of N, it seems that single fault correction should be obtainable at a cost of $\langle \log_2 N \rangle$* extra switches in excess of the CPCU(N) count. In particular we can exhibit specific networks that correct one fault and have switch counts as indicated in Table IV-2. In each case the number of switches shown in Table IV-2 is

Table IV-2

NUMBER OF CELLS IN SINGLE STUCK-FUNCTION CORRECTING CPCU NETWORKS

Leads	Switches in Redundant Network	CPCU(N)
2	2	1
3	5	3
4	7	5
5	11	8
6	14	11

* The $\langle x \rangle$ is the next integer larger than x.

exactly $\langle \log_2 N \rangle$ larger than the corresponding value of CPCU(N). In the cases N = 2, 3 and 4, it is reasonably certain that these realizations are the minimal ones that exhibit the single-fault correcting property. An example of a single-fault correcting network for N = 4 is illustrated in Fig. IV-6(c).

A considerable number of network forms and iterative combining rules were studied during attempts to establish the minimum cost of fault correction. Several time-shared computer programs were written that allow heuristic tinkering with networks, i.e., the program BAD-ONE that verifies whether a proposed network is indeed a full permuter if one cell has a stuck-function fault. As a result of all this experimentation, one feels impelled to make the following conjecture:

> The cost of protection against (correction of) p faults in a CPCU(N) network is no more than the difference in cost between CPCU(N) network and a CPCU(N + p) network. That is, the cost of correcting each additional fault, say fault i, is smaller than $\langle \log_2(N + i) \rangle$.

If this conjecture is indeed true, then there would exist permutation networks of arbitrarily high degree of reliability whose cell-count would not exceed $K \cdot N_1(N)$, where K is a constant related to the probability of a switch failure.

b. An Alternative Single Stuck-Function Correcting Construction

A different and slightly more economical [than the TDT(N) device] method of providing single-fault protection in CPCU(N) networks stems from the observation that the construction of Fig. IV-4(b) can tolerate one cell failure in any peripheral cell if an extra cell, serving outputs Y_1 and Y_N column, is retained rather than deleted. The reason for deleting this cell in the first place was that exactly one peripheral cell is unnecessary in the nonredundant case. Since all of the peripheral switches are exactly equivalent in function, it makes no difference which one we delete. Hence, by retaining all of them we can ignore exactly one failure in any of them. If each of the "internal" permutation networks that implement the construction of Fig. IV-4(b) are augmented in the same way

by replacing the deleted peripheral switch of the CPCU(N) configuration, then we obtain a network, call it $S_1(N)$, that can tolerate one switch failure anywhere. An example of such a network, namely $S_1(4)$, is shown in Fig. IV-7. We notice first that the number of cells, k = 8 so that the network is not minimal [see Fig. IV-6(c)]. However, on counting the redundant cells required when $N = 2^r$, we find that the extra cells are exactly N - 1 in number. This means that in the special case $N = 2^r$ the networks $S_1(N)$ are more economical than those produced by annexing a TDT(N) network to a CPCU(N) network. Recall that this required about 3N/2 extra switches.

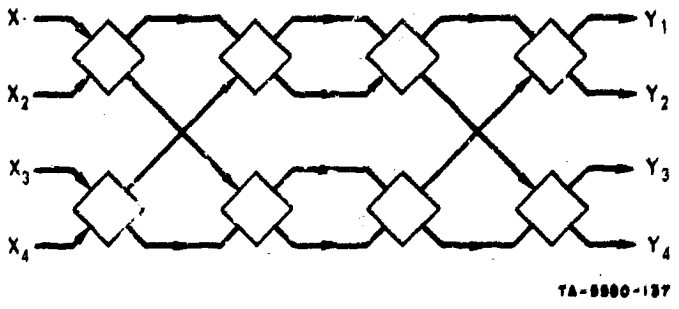


FIG. IV-7 NON-MINIMAL REDUNDANT 4-PERMUTER, SINGLE STUCK-FUNCTION CORRECTING

c. Correction of Bad-Output Fault Types

Considering the effect of a single fault of this type on an otherwise full permutation network, it becomes apparent that exactly one output lead will receive an incorrect signal in response to the applied inputs. The situation can be remedied by adding one extra lead to the N-permutation network if we can also be certain that the input signals are applied to, and the output signals derived from, the correct subset of N leads. One way of accomplishing this is illustrated in Fig. IV-8.* The figure is drawn for the case N = 4, but the method is perfectly

* Clearly a bad-output failure on a cell immediately preceding a network can never be accommodated for. In this case the appropriate byte is disabled.

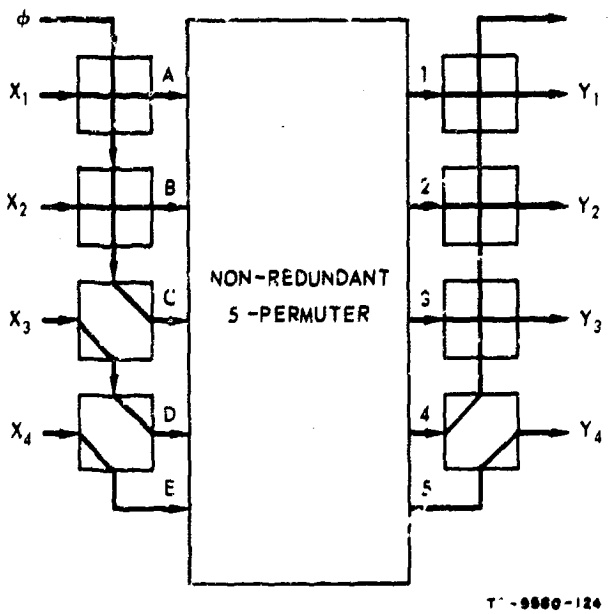


FIG. IV-8 NETWORK FOR CORRECTING BAD-OUTPUT FAULTS

general. To permute N leads, we employ a nonredundant N + 1 permuter flanked on input and output with a ladder network having N switches. If the N + 1 permuter has a bad cell lead, this will show up as a failure of one of the signals on leads A, B, C, D or E of the internal N + 1 permuter to arrive at its specified output. By setting the switches of the ladder network in the obvious manner, the fault can be corrected, as illustrated in Fig. IV-8 for the case wherein input C does not arrive correctly at internal output 4.

The cost of correcting one failure by the method of Fig. IV-8 is 2N switches for the ladder networks plus $\langle \log_2 (N + 1) \rangle$ extra switches [assuming the CPCU(N) construction of Fig. IV-4(b)] to implement the N + 1 permuter rather than the N permuter. This cost, $\approx 2N + \log_2 N$, is asymptotically negligible compared with the cost of a CPCU(N) network for large N. It is obvious that the foregoing construction can be extended to correct multiple faults of the bad-output type.

C. Commutation Networks for Complete Permutation--Incomplete Utilization

We are assuming here that the commutation network is to serve N inputs and N outputs [similar to the function of the CPCU(N) network], but in this case it is only necessary to provide simultaneous connections between $m(m < N)$ inputs and outputs. Such a commutation function, referred to as embodying complete permutation--incomplete utilization, is denoted as CPIU(N,m). It is desired to specify a network that is more economical in terms of cell-count and/or is easier to set up than a CPCU(N) network which, of course, also achieves the CPIU(N,m) function.

It is easy to see that a CPIU(N,m) network must contain enough two-state cells to specify $\binom{N}{m}^2 (m!)$ possible permutations; thus, $N_2(N,m)$, the number of cells in the network, is given by

$$N_2(N,m) \geq \log_2 \left[\binom{N}{m}^2 (m!) \right]$$

or

$$N_2(N,m) \geq 2 \log_2 \binom{N}{m} + N_1(m) \quad .$$

The above formulas suggest that the CPIU(N,m) function could be realized, as depicted in Fig. IV-9, by a network composed of a CPCU(m) block (m-permuter) sandwiched between two combination networks.

It is assumed that a combination network serving m inputs and N outputs, denoted as a COM(m,N) network, is to have the capability of connecting

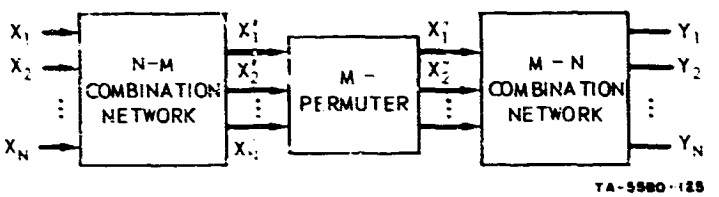


FIG. IV-9 SCHEMATIC REPPRESENTATION OF DECOMPOSITION OF A COMPLETE PERMUTATION INCOMPLETE UTILIZATION NETWORK

the set of m inputs onto any specified set of m output leads,* without regard to the order of these signals on the outputs. A similar definition applies to a COM(n,m) network where the number of inputs is assumed to exceed the number of outputs.

The number of cells, $N_c(N,m)$, required for an m,N (or an N,m) combination network is given by:

$$N_c(N,m) \geq \log_2 \binom{N}{m} = \log_2 \left[\frac{N!}{m!(N-m)!} \right] .$$

Asymptotically, from Stirling's formula, we find that a network composed of N - 1 cells should be sufficient to perform the combination function.

We have <u>not</u> found combination networks, composed of the 2-input basic cell which approach this N - 1 cell bound, as closely as the CPCU(N) networks approach the $(\log_2 N!)$ bound. However, it is not difficult to specify a COM(N,m) network that requires only N two-state cells, but wherein each cell contains m + 1 inputs.

Consider the two-state cell depicted in Fig. IV-10, with m horizontal inputs (m = 3 for the case shown), I_1, I_2, ..., I_m, m horizontal outputs O_1, O_2, ..., O_m, one vertical input, X_i, and one dummy (unused) vertical

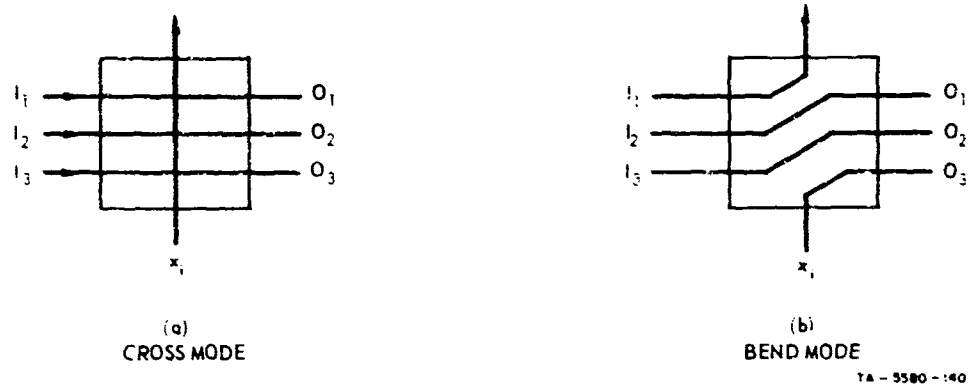


FIG. IV-10 BASIC CELL WITH AUGMENTED SET OF INPUTS

* As indicated in Sec. IV-B-1, if a network is commutating parallel data channels, each call input is in reality a bundle of many wires.

output. In the "cross" mode, the three horizontal inputs are transferred unaltered to the three horizontal outputs. In the bend mode we effect the transformation $X_i \rightarrow O_m$, $I_m \rightarrow O_{m-1}$, ..., $I_2 \rightarrow O_1$. An m,N combination network is easily synthesized as a cascade, containing N of these augmented input cells, as shown in Fig. IV-11 for the case m = 3, N = 6. The appropriate cell modes are shown for the case where it is desired to connect the inputs X_2, X_1, and X_5 to the three outputs, Y_1, Y_2, Y_3, which are the horizontal outputs of the last cell in the cascade. It is noted that the first cell in the cascade actually requires no horizontal inputs, the second cell only 1 horizontal input, ..., the mth cell only m - 1 horizontal inputs. However, if we consider the approximate cost of a cell to be proportional to the number of terminals served, then the cost of the combinational realization of Fig. IV-11 is of the order of mN. We can find realizations that require a number of cells in excess of N, yet exhibit a cost measure significantly less than the network previously described.

We will recursively synthesize a COM(m,N) network that is composed of the basic two-input cell, using a procedure suggestive of the one described in Sec. IV-B-1. Consider the COM(m,N) network represented by Fig. IV-12, where it is assumed that $2|m$ and $2|N$. The subnetworks, Q_1 and Q_2, are themselves combination networks, each with half of the number of inputs of the total network. This arrangement requires a number, $N_2(m,N)$, of cells which satisfies:

$$N_2(m,N) = 2N_2(m/2, N/2) + \frac{N}{2} - 1 \quad .$$

Solution of this recursion for $N = 2^r$, $m = 2^{r-1}$, using $N_2(1,2) = 1$*) yields

$$N_2^2(2^{r-1},2^r) = 2^{r-1} + (r - 2)2^{r-1} + 1$$

---

* It is clear that a single two-input cell, where one of the inputs is not used, is a COM(1,2) network.

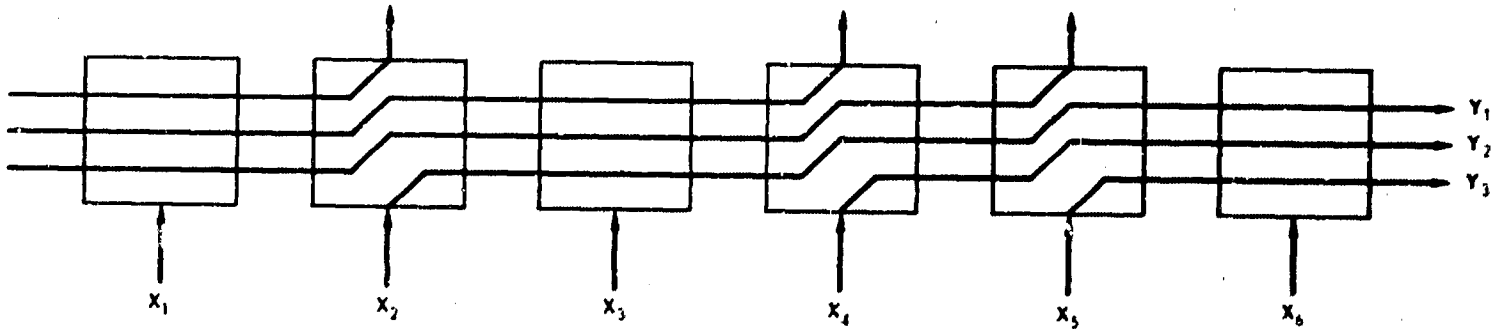


FIG. IV-11 N, m COMBINATION NETWORK

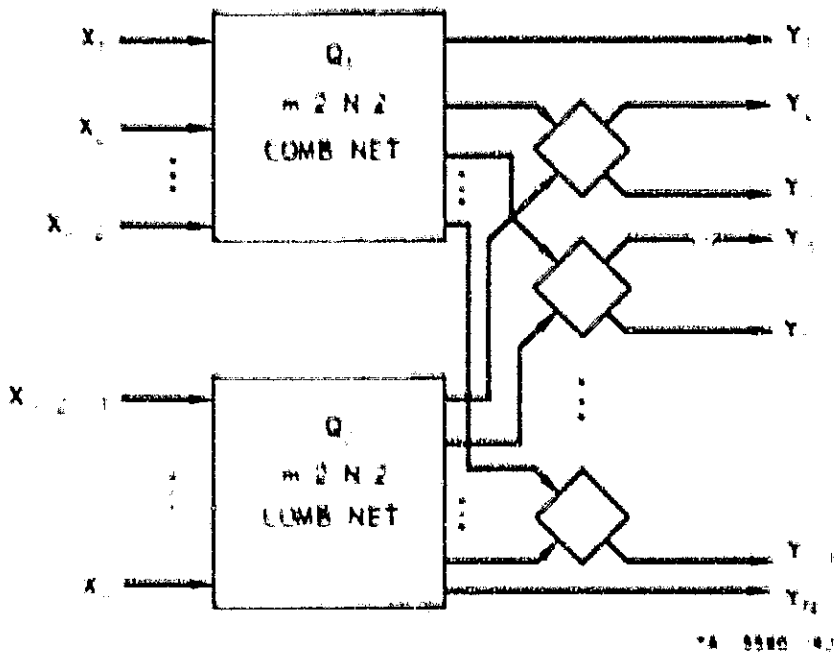


FIG. IV-12 RECURSIVE APPROACH TO m - N COMBINATION NETWORK

or

$$N_2(2^{r-1},\ 2^r) = \frac{N}{2}\log_2(N) - \frac{N}{2} + 1 \quad .$$

A similar expression can be derived for the case wherein m, N are not powers of 2.

A constructive proof that this network is capable of developing a path from the m inputs to an arbitrarily selected set of m outputs is as follows. Let the inputs and outputs be labeled $X_1, X_2, \ldots, X_m$, and $Y_1, Y_2, \ldots, Y_N$ as indicated in Fig. IV-12. We will start by indicating the appropriate modes for the (N/2) - 1 (output) cells serving outputs $Y_2, \ldots, Y_{N-1}$, so that for an arbitrary selection of m outputs, exactly m/2 of these outputs are connected to Q_1 and m/2 to Q_2. If an output cell serves two selected outputs, it can be arbitrarily set to either mode. The output cells which serve the remaining set of selected outputs are then set to the appropriate mode so that the first of these outputs (including possibly Y_1) is connected to Q_1, the second connected to Q_2, etc.

The entire procedure is now repeated for each of the networks Q_1 and Q_2, etc., until the entire network has been set up. The network for

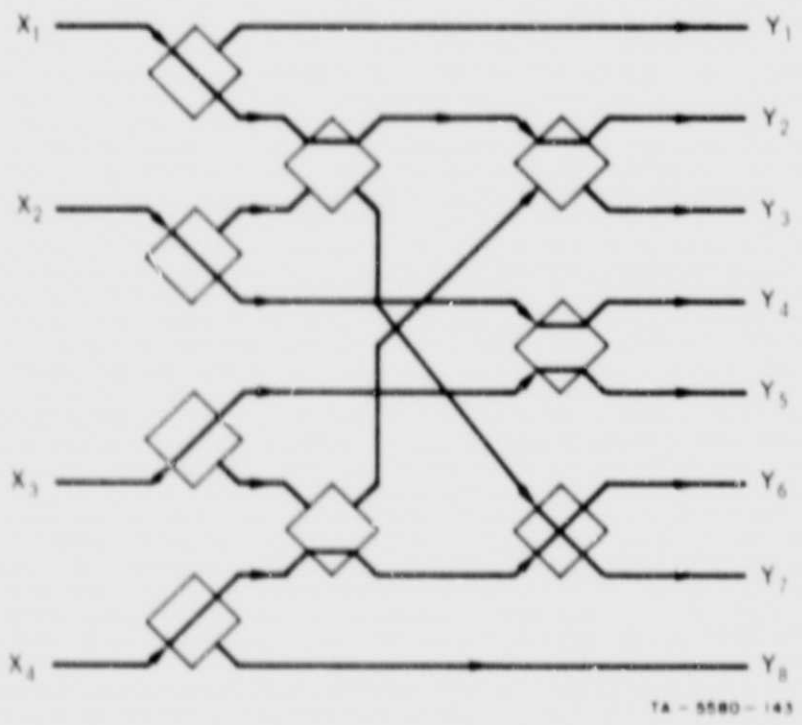


FIG. IV-13 4-8 COMBINATION NETWORK

m = 4, N = 8, is shown in Fig. IV-13, where the cell functions are indicated for the selected output set Y_2, Y_4, Y_5, Y_6.

It is interesting to investigate techniques for incorporating redundancy into these combination networks such that they can continue to effect a given assignment in the presence of cell failures. Considering the stuck-function failure, a simple technique can be specified, for single failures of this type, similar to the method described in Sec-B-3(b). Referring to Fig. IV-12, we note that an output cell was not required to serve outputs Y_1 and Y_N, and a corresponding cell was omitted from each internal subnetwork. If we insert this cell in each case, and in addition duplicate the cells serving the input leads, we will have added N - 1 cells and produced a network that is tolerant to single stuck-function failures. In Fig. IV-14 we display such a redundant 4 - 8 combination network and indicate the appropriate cell modes, so as to connect the inputs to Y_1, Y_3 wherein cell S is assumed stuck in the cross mode.

For the case of bad-output failures, a technique similar to that described in Sec. IV-B-3(c) can be applied. The technique for a single failure in the combination network case would require a COM(m + 1, N + 1)

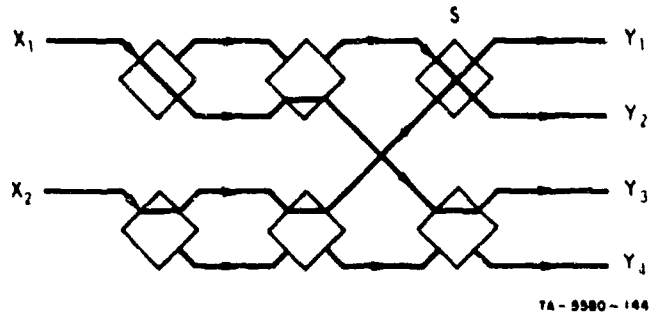


FIG. IV-14 REDUNDANT 4-8 COMBINATION NETWORK FOR CORRECTION OF SINGLE STUCK-FUNCTION FAILURES

network flanked by ladder containing m cells on the input and a ladder containing N cells on the output. The technique generalizes easily to accommodate to multiple failures.

D. Commutation Networks for Incomplete Permutation--Nonorder Preserving

It may be recalled that incomplete permutation--nonorder-preserving networks are required to establish connection paths between a set of inputs and outputs, where only a subset of the inputs and outputs are required for a particular task, and the spatial ordering of the signals at the input does not have to be retained at the output. One application of these networks, which has been described, is concerned with the transfer of data between registers where a redundant set of registers is specified.*

We note, somewhat trivially, that a COM(m,N) network would function as an incomplete permutation--nonorder-preserving network serving a nonredundant set of m inputs and a set of N outputs of which N-m are redundant. In this section we will be concerned with networks for the incomplete permutation--nonorder-preserving function, IPNOP(r,m), wherein there are r inputs and r outputs, and it is necessary to connect m input-output

* Note that this application is quite different from the case wherein we are concerned with the transfer of information between registers that have redundant bits. In the latter case we would require an incomplete permutation--order-preserving network.

pairs together without regard for spatial order. For example, if r = 6, m = 3 and we distinguish inputs X_1, X_3, X_4 and outputs Y_2, Y_4, Y_6, then the network is functioning properly if it would establish <u>one</u> of the following assignment sets, $[X_1 \rightarrow Y_4, X_3 \rightarrow Y_2, X_4 \rightarrow Y_6]$, or $[X_1 \rightarrow Y_6, X_3 \rightarrow Y_2, X_4 \rightarrow Y_4]$, etc.

It can be shown that a lower bound on the number of two-state cells required for a IPNOP(r,m) network is between $\log_2 \binom{r}{m}$, and $2 \log_2 \binom{r}{m}$.[‡] In Section IV-E, which is concerned with the order preserving case, we will describe an incomplete permutation--order-preserving network composed of 2r two-state cells that can, of course, function is a IPNOP network. However, each cell in the network must serve up to m + 1 inputs, providing an overall network cost that approaches 2r(m + 1). We will now describe an IPNOP(r,m) network that requires more than 2r cells, yet exhibits a cost measure significantly less than 2r(m + 1).

Consider the network shown in Fig. IV-15, which we will demonstrate yields recursively an IPNOP(r,m) network, where R_1 and R_2 are each IPNOP(r/2, m/2) networks. This arrangement yields a number of two-state cells, $N_3(r,m)$, which satisfy

$$N_3(r,m) = 2N_3(r/2, m/2) + r - 2 \quad .$$

Solution of this recursion for $4 = 2^k$, $m = 2^{k-1}$ and, using $N_3(2,1) = 1$,[†] gives

$$N_3(2^k, 2^{k-1}) = 2^k(k - 1) - 2^{k-1} + 2$$

[‡] The proof of this bound is deferred until the final report under this contract. It appears for this case that the lower value is the tighter bound.

[†] It is clear that a single two-input, two-output cell is an IPNOP(2,1) network. It is also a IPOP(2,1) network, a property which will be exploited in the succeeding section.

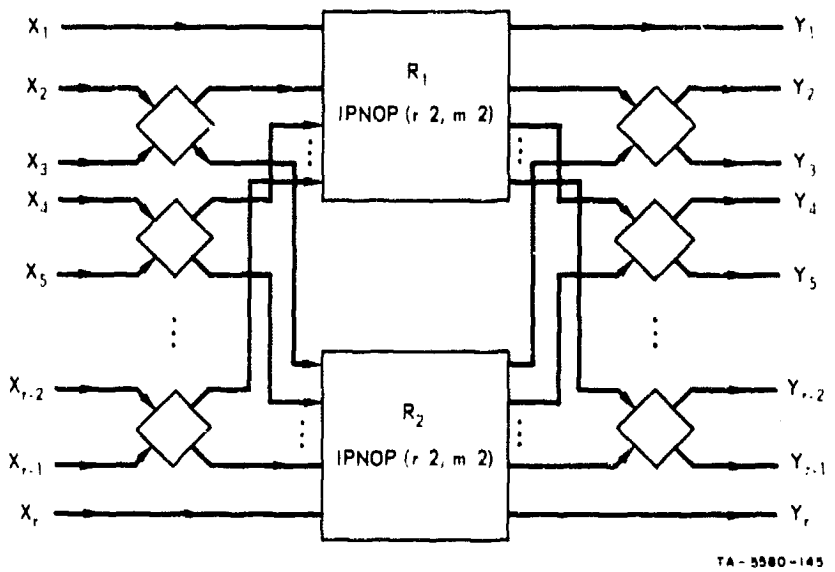


FIG. IV-15 RECURSIVE APPROACH TO INCOMPLETE PERMUTATION — NONORDER PRESERVING NETWORK

or

$$N_3(r, r/2) = r \log_2 r - \frac{3}{2}r + 2 \quad .$$

Although we considered only the case $m = r/2 = 2^{k-1}$, the recursive technique is quite general and will yield a network corresponding to arbitrary parameters, r, m.

A constructive proof that this network is capable of finding a mate for each input and output contained in an arbitrary set of m inputs and m outputs is quite similar to the proof provided in Sec. IV-C for the combination network. For the network of Fig. IV-15 it is apparent that the peripheral cells immediately serving inputs $X_2, \ldots, X_{r-1}$, and the cells serving outputs $Y_1, \ldots, Y_r$ can be set so that exactly half of m distinguished inputs are directed to R_1 and half are directed to R_2; an identical requirement is satisfied for the m distinguished output leads. The entire procedure is repeated for each of the networks R_1 and R_2, etc., until the entire network has been set up.

Procedures, similar to those described in Secs. IV-B-3(b) and IV-B-3(c), can be applied to the IPNOP(r,m) network, so that the network is tolerant to stuck-function and bad-output type faults. For the stuck-function case we note, by referring to Fig. IV-15, that a single cell is missing on the input and output portions of each subnetwork. The insertion of this cell at each level in the network will yield a network tolerant to single stuck-function cell failures.

E. Commutation Networks for Incomplete Permutation--Order Preserving

It may be recalled that the memory modules, arithmetic logic units, and simple processor and control units can be realized as a cascade of identical byte slices. These modules can continue to function, upon the occurrence of failures in slices, if several spare slices are provided, and if a commutation network is provided to route the signals between operating slices. The function of such a commutation network, denoted as an incomplete permutation--order-preserving network, IPOP(r,m), is to set up connecting paths between an arbitrary set of m inputs and an arbitrary set of m outputs, both sets of which are subsets of the r inputs and r outputs, $r > m$, so that the signal order is the same at both input and output.

Similar to the nonorder-preserving case, it can be shown that a lower bound on the number of two-state cells required is $\log_2 \left(\frac{r}{m}\right)$ and $2 \log_2 \left(\frac{r}{m}\right)$, although for the order-preserving case, the upper value appears to be tighter. It is possible to realize the IPOP(r,m) function in a network composed of 2r two-state cells, where each cell contains m + 1 inputs. It is seen that this network approximately satisfies the $2 \log_2 \left(\frac{r}{m}\right)$ bound for the case $m = r/2$ since

$$\lim_{r \to \infty} \log_2 \left(\begin{pmatrix} r \\ [\frac{r}{2}] \end{pmatrix} \right) = 2r - 2 \quad .$$

The basic cell is the type shown in Fig. IV-10, and the network is displayed in Fig. IV-16 for the case $r = 6$, $m = 3$: the modes of the cells

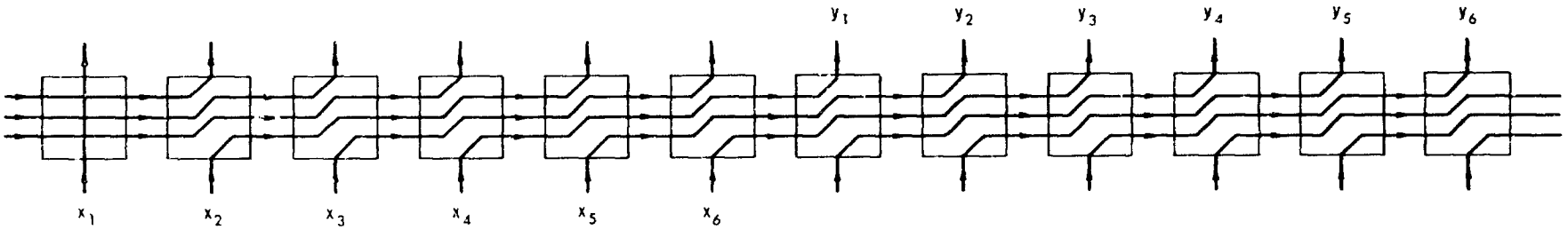


FIG. IV-16 AN INCOMPLETE PERMUTATION — ORDER PRESERVING NETWORK

are such as to realize order-preserving connections between inputs X_2, X_3, X_5 and outputs Y_4, Y_5, Y_6. Even though the number of horizontal inputs served by the first m - 1 cells in the cascade and the number of horizontal outputs served by the last m - 1 cells in the cascade can both be reduced, the cost of this network is of the order of 2mr, a considerable cost. Similar to the situation involving the other commutation functions, the cost of the realization is significantly reduced if the two-input cell is used as the basic primitive block.

A network composed of two-input cells, which realizes the IPOP(r,m) function, is identical to the network (Fig. IV-15) for the nonorder-preserving case. We have redrawn the network of Fig. IV-15 as Fig. IV-17 for the case r = 8, m = 4. It may be recalled that for the nonorder-preserving case the function of the (r - 1) output cells and the r - 1 input cells was to connect the distinguished m input and m output leads to the two subnetworks, R_1 and R_2, such that exactly m/2 distinguished inputs and m/2 outputs were connected to each of the networks R_1 and R_2. For the nonorder-preserving case the input and output cells could be set independently; such is not the case if the network is to be used as an order preserving network.

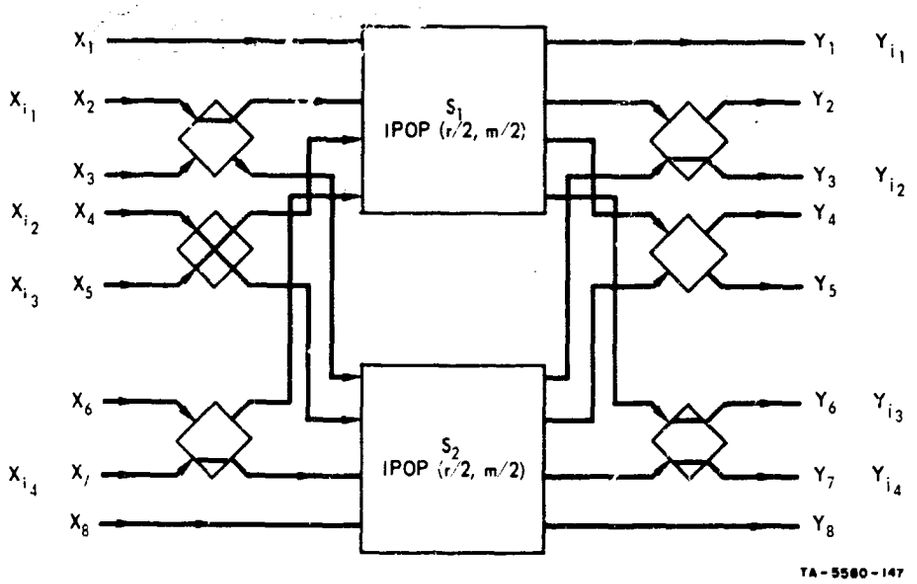


FIG. IV-17 RECURSIVE APPROACH TO INCOMPLETE PERMUTATION — ORDER PRESERVING NETWORK

The following procedure, for the order-preserving case will indicate the proper mode of each cell of the network of Fig. IV-17, for a given set of m inputs and outputs; and, hence, will prove that the network can function as an order-preserving network. Let the distinguished set of m inputs be $X_{i_1}, X_{i_2}, \ldots, X_{i_m}$, where $i_\alpha > i_\beta$ if $\alpha > \beta$, and the m outputs be $Y_{j_1}, Y_{j_2}, \ldots, Y_{j_m}$, where $j_\alpha > j_\beta$ if $\alpha > \beta$. Consider each of the m inputs as residing in one of two disjoint groups. Group A_I contains those inputs that do not share an input cell with another distinguished input, and group B_I contains those inputs that do share an input cell. We will similarly define groups A_O and B_O for the distinguished outputs. The goal is to assign X_{i_1} and Y_{j_1} to the same subnetwork (S_1 or S_2), X_{i_2} and Y_{j_2} to the same subnetwork, etc., and the procedure is as follows.

Assign X_{i_1} and Y_{j_1} to network S_1, by appropriately setting the pertinent input and output cells (except for the case where $X_{i_1} = X_1$ and/or $Y_{j_1} = Y_1$, in which case the assignment to S_1 is automatic). Then assign X_{i_2} and Y_{j_2} to network S_2; if X_{i_1} and X_{i_2} are in group B_I, and/or Y_{j_1} and Y_{j_2} are in group B_O, the assignment to S_2 is automatic. Next, assign X_{i_3} and Y_{j_3} to S_1, etc., until all of the m distinguished inputs and outputs have been set. This procedure is then applied to set the pertinent output and input cells of the networks S_1 and S_2, etc. It is clear that this assignment procedure can always be carried out. In Fig. IV-17 we show the setting of the input and output cells for the case $X_{i_1} = X_2$, $X_{i_2} = X_4$, $X_{i_3} = X_5$, $X_{i_4} = X_7$ and $Y_{j_1} = Y_1$, $Y_{j_2} = Y_3$, $Y_{j_3} = T_6$, $Y_{j_4} = Y_7$.

The techniques for providing failure tolerant IPOP networks are not discussed in this section since they are quite similar to the techniques described in Secs. IV-B-3(b) and IV-B-3(c). We note that a cell failure in the IPOP network (two-input cell type) can disable no more than two byte slices each for the input and output. Since it is assumed that redundant slices are provided, it is possible that a nonredundant network would be used, and when cell failures are detected, the slices that could not be served by the network would be discarded.

F. Commutation Networks for "Shorting"

In Sec. IV-E we described networks that, for a redundant byte-sliced network, can serve to route external data between the operating slices of distinct networks (e.g., between an SP and ALU). It was noted[1] that internal data (e.g., control and carry information) must be routed between the stages of the byte-sliced cascade. If a state (or slice) has failed, then the internal data intended for that stage, which clearly comes from its immediate predecessor or successor, must be shorted around that failed stage. If this shorting process is not accomplished reliably, then the entire network will be disabled.

The shorting function is quite naturally achieved with the two-input basic cell, as illustrated in Fig. IV-18. For simplicity, only a signal flow to the right has been indicated although it is clear that the network could be modified to handle bi-directional flow. We have shown the appropriate cell modes so that byte slice 2 and byte slices 5 and 6 are shorted out. We note that the network could recover from a single component failure within a cell, which results in either the stuck-function failure or the bad-output failure. However, a more severe cell failure which results in, for example, a permanent logical zero signal on both outputs of a cell would clearly disable the network, i.e., interrupt the signal flow.

Such a failure, which could only result from two component failures within a cell, could be accommodated for by the redundant shorting network of Fig. IV-19. Also indicated are the appropriate modes for cells S_1, S_1', S_2, S_2' such that byte-slice 1 is shorted out, i.e., the output from slice 0 is directed to the input terminal of slice 2. We have also shown the appropriate modes for cells S_3, S_3', S_4 such that the network continues to function although both outputs of S_4' are faulty. In this case byte-slice 3 cannot be used, but the signal flow is not interrupted. Similarly we have shown how the network accommodates to a double-output failure in S_5 in which case slice 5 is bypassed. This technique can be clearly extended to handle failures of greater multiplicity.

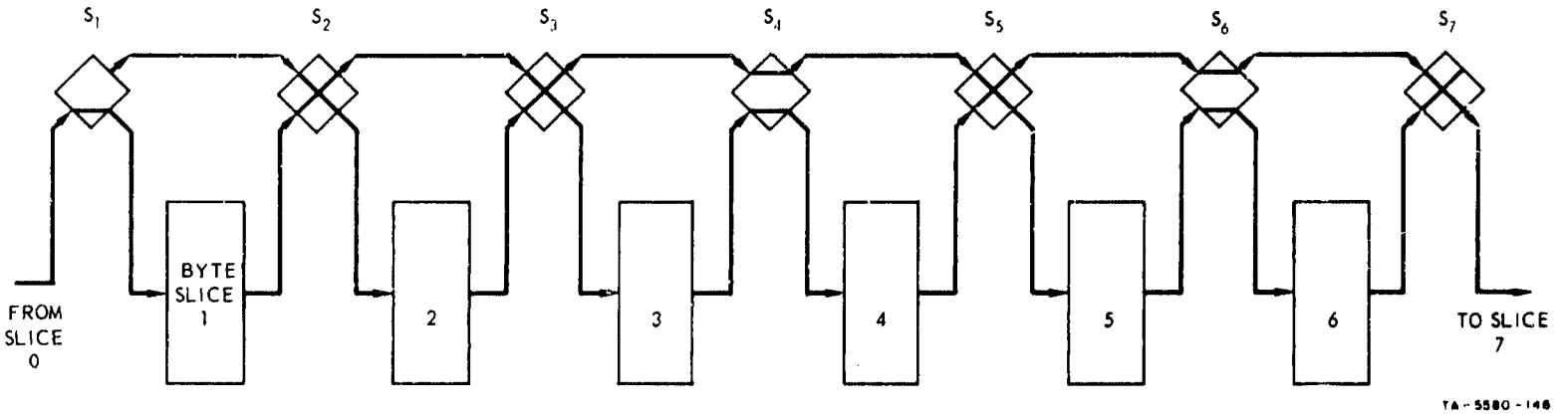


FIG. IV-18 "SHORTING" NETWORK

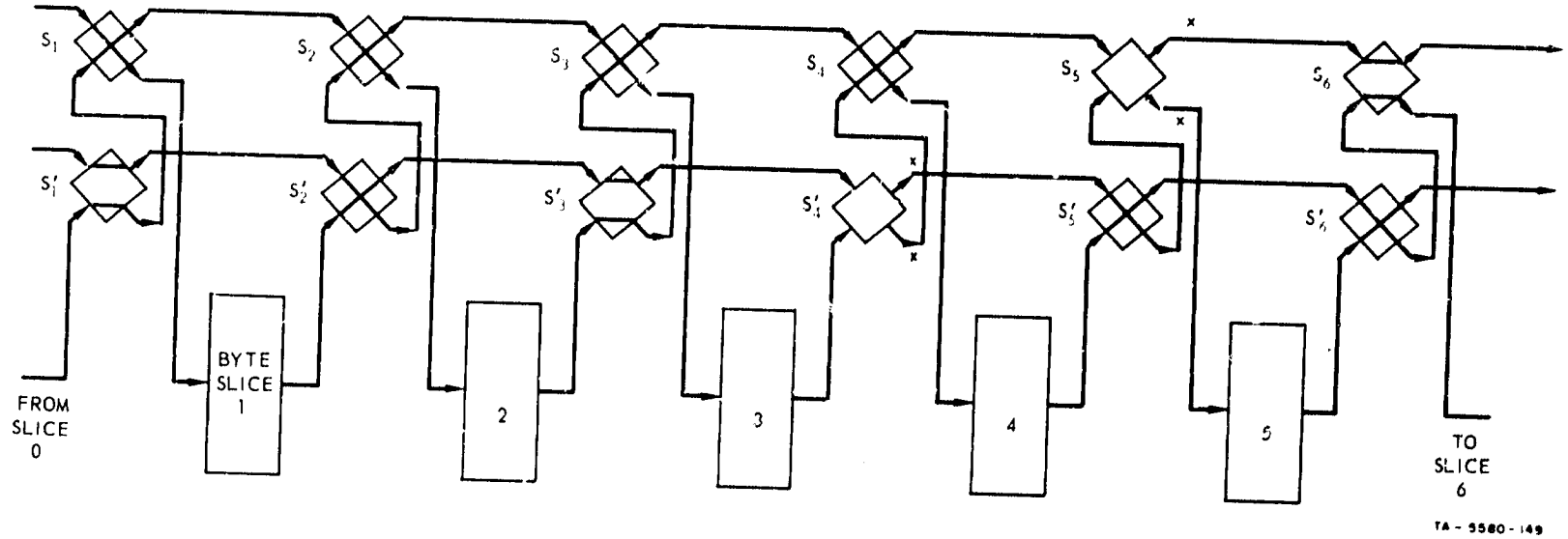


FIG. IV-19 REDUNDANT SHORTING NETWORK

G. Summary

In this chapter we have studied in detail the logical design of networks that could perform the various data switching or commutation required in a multiprocessor organization where the various modules are repairable. It is assumed that the memory, arithmetic logic, and possibly the simple processor and control modules are realized in a byte-sliced manner--a realization that has been demonstrated to be practical. It is felt that the designs we have presented, based upon the primitive two-input, two-output reversing cell represent adequate engineering solutions to all of the commutation problems posed, although some theoretical minimization problems still remain. These problems relate to minimum cost designs for the complete- and incomplete-permutation functions considering both the nonredundant realizations and the realizations that are tolerant to cell failures.

V ULTRARELIABLE PROGRAMMING

In this section the problem of constructing ultrareliable computer programs is treated. The term "ultrareliable" in this context refers to a program that not only operates completely flawlessly in normal circumstances, but one which is insensitive to faults that are introduced through the input data stream or certain hardware failures. The approach used here is to classify the common reasons for faulty programs, and discuss methods for the prevention of these faults, the detection of failures arising from such faults, and the recovery within the computer system from detected failures. Although the principal interest of this section is computer software, because of the intimate relation of computer software to computer hardware, in many cases the appropriate means for attaining reliable programs will be hardware oriented or will be a combination of hardware and software techniques.

In Sec. A the classification of common software faults is presented. Each class is treated separately in Secs. B-D, and Sec. E contains a short summary and a list of several unresolved problems that are identified in the course of this discussion.

A. Classification of Program Faults

For purposes of exposition, we define a fault to be a program characteristic that can cause a program to execute improperly under some set of conditions that may depend on program state, input data, or timing. An instance in which a program executes improperly is said to be a failure. In this section, common program faults are grouped into one of three categories according to qualities that determine methods for eliminating the faults. The categories can be briefly described as faults due to problems of data analysis, program checkout, or the execution of a program in a context that is outside of the scope of its validation and definition.

These faults can be attributed to human mistakes in the composition of the program or to hardware faults in, for example, a back-up memory where a program is stored. More specifically the categories are:

Type I. Program algorithm is essentially correct, but program produces inaccurate results or fails to terminate because of problems in numerical analysis.

Type II. Program contains bugs, i.e., has faults such that it fails to perform processes according to specifications.

Type III. Program is completely correct for its scope of activities, but fails when operating outside its scope.

Because the statements above are stated rather broadly, the categorization immediately allows us to make some statements about a general methodology for achieving reliable programs.

Type I faults are principally due to failure of an algorithm to account for cumulative effects of round-off and truncation errors, or operates on a range of data for which the algorithm is unstable.* The methodology to be used for this class of faults is oriented to problems in the representation and manipulation of numerical data in a computer. Rather than attempt to address this section to the entire field of numerical analysis, we shall merely call attention to the need for such analysis and devote the greatest portion of the discussion to the design of hardware that will alleviate many problems of numerical analysis.

The methodology to be used to eliminate Type II faults, the programming bugs, is slanted to the use of redundancy in program specification to permit software support programs to aid the creation and check-out of reliable programs. Program bugs can arise from many sources and all are

---

* At initial observation numerical appears to be strictly a software problem which is solved once, i.e., when the problem is written, but within the graceful degradation concept we are proposing, it is imperative to detect instabilities in algorithms caused by an insufficient quantity of equipment remaining available for program execution.

susceptible to some checks. Transcription errors can be caught by redundancy in the language, logical errors by the use of aids like decision tables, and blunders by consistency and completeness checks that are programmed to be independent of the algorithm that they check.

Type III faults are intended as a catch-all category. Failures that arise either from undetected Type I or II faults or from minor transient hardware failures generally appear in programs that are otherwise error-free. It is common practice to validate programs by checking their behavior with test data that lies in their scope of validation, i.e., for the if conditions of hypothetical cases. Few programs are written to work properly only if the data and the state of the program are confined to the scope of validation. To guard against Type III faults, we use the methodology of if and only if programming. This is the practice of using extensive checks, both software and hardware, to validate all input data and, when possible internally generated data in order to guarantee that they are in the range of definition and remain so during the course of a computation.

B. Faults Arising from Numerical Analysis

1. The Need for Analysis

Most of the problems of numerical analysis that give rise to computation failures are due to the finiteness of the representation of numbers in a computer. If, in practice, numerical representations can be made arbitrarily long, representation errors can be made arbitrarily small. Nevertheless, for practical reasons, numerical data are fitted into fixed-length fields that are deemed to be sufficiently long to give accurate results in most cases. The length, which varies from machine to machine, is typically between 24 and 56 bits long for floating-point mantissas. A calculation, i.e., the implementation and execution of an algorithm on a specific machine, must be subjected to thorough numerical analysis to guarantee that it will produce results of sufficient accuracy.

To borrow an example from Ccdy,[19] let us examine the computation of the mean of two numbers and use the formula

$$M = (x_1 + x_2)/2 \quad .$$

If the representation of x_1 and x_2 is a floating point, and the base is other than 2, then in a binary computer it is possible to compute a mean M that is less than either x_1 or x_2. For example, consider the computation using radix 10 arithmetic with two digit precision. Let $x_1 = 51$, $x_2 = 52$, and note that $51 \cdot 10^0 + 52 \cdot 10^0 = 103 \cdot 10^0$, which, to two places accuracy, is $10 \cdot 10^1$. Division by two yields $50 \cdot 10^0$, which is less than either operand.

There are two serious effects of this type of fault. The most obvious is in the introduction of a small error in the least significant digit that is slightly greater than the apparent accuracy of the computation. A more subtle effect of the error is to place the result outside the theoretical range of possible answers. The latter effect could lead to instability of a computation.

A second example of the pitfalls of computation is given by Neely,[20] who illustrates several alternate computations for the mean, standard deviation, and correlation coefficient calculation on typical statistical data. Although the several alternatives are algebraically equivalent, the computations give widely disparate results. In his example, the most direct computations generate answers that are among the least accurate, and the simple expedient of carrying out the calculation in double precision results in answers that are among the most accurate.

The point of these examples is to demonstrate the need for numerical analysis to aid in the development and analysis of computations for particular computer systems. It is vital that the analysis be done in the context of the computer on which the algorithm is to be executed, of course, because it is precisely the idiosyncratic behavior of computer arithmetic units that requires the detailed numerical analysis. The grosser characteristics of numerical processing are inherently consistent from machine to machine.

Because the numerical analysis is so broad a subject, we cannot be more explicit than to issue a caveat to the programmer to take into account the problems of numerical analysis. However, there are other general guidelines that fall in the area of the computer design which we can explore here. In particular, the hardware can be designed so as to minimize the errors of numerical representations, and specific steps can be taken for the detection of computational errors, and for the programmatic recovery from errors when they are detected. We pursue these questions separately in the following subsections.

2. Design of Floating-Point Hardware to Aid Numerical Analysis

Two differing viewpoints have come to be reflected in the design of floating-point hardware. The first focuses on obtaining the greatest possible precision of an operation and usually depends on normalized arithmetic. The second viewpoint, which is somewhat opposed to the first, is concerned primarily with obtaining results that are known to be significant at the possible expense of precision. The latter is implemented primarily in unnormalized arithmetic. Both viewpoints are compatible with reliable computing, because the best estimate of the true value of an answer can be obtained through the use of normalized operations, whereas unnormalized operations can be used to give an estimate of the range of possible values that may contain the true answer. In this section we discuss the design of floating-point hardware for both normalized and unnormalized operations.

We first consider normalized floating-point operations. In the previous section, the calculation of a mean yields an erroneous result because significant data is shifted off the right hand portion of an intermediate result and replaced by significant 0's. Even if the operation is normalized floating-point addition, it is possible to lose significance as shown in the example when the floating-point radix is greater than 2. When the radix is not binary, then the representation of a normalized quantity in a binary machine may have leading 0 bits ($10 \cdot 10^{1}$ in the example) and the leading 0's are obtained at the expense of bits lost from the least significant portion of an operand. Hence, in a binary computer, normalized floating-point arithmetic in a radix other than 2 contributes to the inaccuracy of

computation. The number of bits of lost precision is approximately one half the base two logarithm of the floating-point radix. For radix 2, this loss is 1/2 bit, which is equal to the intrinsic loss of precision in the binary representation of real quantities. For larger radices, the loss of significance becomes greater and can become quite noticeable because it introduces a biased error in the representation. Single precision on System/360 computers, for example, is radix 16 with 24 bit mantissas, yielding a significance loss of two bits in 24 (a precision of one part in $6 \cdot 10^6$). This is very low for general purpose computation and must be used with caution.

To obtain the most reliable results, it is clear that radix 2 yields the greatest precision for a fixed mantissa length in a binary computer. There is a trade-off involved in using radix 2 arithmetic because the increased precision of mantissas comes at the cost of increased length of exponent. Larger radices appear to be attractive because for each bit of exponent that is saved, only 1/2 bit of significance of mantissa is lost. However, because larger radices introduce a biased error, and because the apparent economy of representation, using large radices, is only one or two bits for radices that are reasonable to implement, it is recommended that radix 2 arithmetic be used for normalized floating-point operations, especially where reliability is an important factor.

There are several details in the manipulation of numerical quantities that require examination. It is straightforward to construct a repertoire of normalized arithmetic operations that consistently yield the highest possible precision. Nevertheless, the most important details are given here because so few computers have incorporated all of these details into their hardware. In the material that follows it will be assumed that the floating-point operations are radix 2.

Floating-point addition and subtraction operations introduce significance loss when an operand is shifted during exponent adjustment. If we assume that operands are normalized, then the smaller of a pair of operands must be right-shifted until its exponent is equal to that of the larger operand. Hence, significant digits are shifted off the right of an operand prior to forming a sum.

A single guard bit at the right-hand end of an adder can be used to ensure that the smaller operand is rounded rather than truncated prior to addition. Both truncation and rounding introduce errors in significance, but rounding errors are preferable because they tend to be unbiased.

Multiplication in floating-point representation can sometimes yield unnormalized products, even when both operands are normalized. Post-normalization of the product involves a left shift of no more than one bit, but that bit should be significant to ensure full precision. Hence, the multiplication hardware must make provision for saving a guard bit from the partial product in case a postnormalization left shift is necessary. Rounding of the final product should occur after postnormalization so that there must be provision for a second guard digit for rounding the product in case postnormalization is necessary.

Accumulators in computers can be constructed with several guard digits so that several intermediate operations can use extended precision operands, then round the numbers to single precision prior to restoring in main memory. In actual practice, many algorithms make use of double-length accumulators for intermediate calculations, following the philosophy that is suggested here. There is a trade-off to be considered here because the number of guard digits that should be kept in an accumulator depends on the relative magnitude of the operands and the number of operations to be performed prior to restoration of data in memory. It may be possible, for example, to obtain satisfactory precision by extending the accumulator by four to six bits, and, thereby, save a second full-length accumulator for other purposes. When the nature of the calculations are known, their characteristics should be taken into account to determine how extended precision operations can best be implemented.

When single precision is inadequate for a computation, all arithmetic operations can be made sufficiently accurate by using multiple precision arithmetic and extended length numerical representations. Some multiple precision operations might be included in the computer instruction repertoire, but provision should be made for performing extended precision operations of arbitrary length by using appropriate software subroutines.

To simplify the multiple precision software, it is essential that the mantissa overflow bit be program-accessible so that overflow bits can be treated as carries from word to word in the extended precision representation of a number. It is also necessary that overflow be signalled after the completion of an operation and that the single precision result of an operation that produces an overflow condition be significant except for the information that is held in the overflow bit. In practice, some computers treat overflow as an error condition and place meaningless information in the mantissa of the result. This places an undue burden on the task of programming the multiple precision operations. Overflow of exponent should be treated in the same manner as overflow of mantissa.

The steps above are suggestions for obtaining the greatest possible precision from floating-point hardware. An alternative viewpoint is to obtain results that are guaranteed to be fully significant. For this purpose the unnormalized mode of floating-point arithmetic has been recommended and is described in detail elsewhere.[21] Unnormalized arithmetic achieves its goal by controlling the postnormalization of results to eliminate the introduction of insignificant data. For addition and subtraction, postnormalization is eliminated completely except when overflow occurs. Multiplication and division use more complex formula to determine the amount of postnormalization. The discussion on the dangers of truncation as opposed to rounding and the need for guard digits applies to unnormalized operations as well as to normalized operations.

3. Detection of Failures Arising from Numerical Analysis

Analysis problems usually lead to one of two types of failures, loss of numerical significance or algorithm instability. We deal with each of these separately.

To detect loss of significance, several competitive methods might be implemented in an ultrareliable system. We have already discussed the use of unnormalized arithmetic briefly, and note that this is attractive for guaranteeing the significance of the result. Calculations can

be performed in both normalized and unnormalized modes so that the best estimate of the true result can be obtained from the normalized answer while the unnormalized answer can be used to indicate the precision of the calculation.

Another scheme is to perform a calculation in two modes in order to define the endpoints of the interval of numbers that contains the true answer. One endpoint is calculated by using basic operations that always truncate their results and by using algorithms that provide a lower bound for the true result. The second mode is identical to the first except that results are always rounded upwards and the algorithm is programmed to provide an upper bound for the true result. Intermediate operands and final results are intervals on the real instead of simple real numbers, hence the name interval arithmetic. A full discussion appears in Sec. V-B-4.

Check algorithms can be used to reveal the accuracy of a result. Linear problems, for example, usually yield a set of residuals that should sum to zero. The actual sum of the residuals provides an estimate on the significance of the result. In many other cases, there is sufficient data available at the end of a calculation that can be used for similar check computations.

Another scheme that has been implemented in some commercial computers is the use of a significance alarm to detect excessive prenormalization or postnormalization shifts during floating-point addition operations. One or more bits per operand can be allocated to store the significance of the operand. A single significance bit can only differentiate between significant and insignificant operands, whereas several significance bits permit several levels of significance to be represented. These bits can be maintained automatically by the hardware. This is seen to be another form of unnormalized representation when the number of significance levels equals the number of bits in the mantissa.

The second topic for this section is that of the detection of instability in numerical algorithms. The problem is to determine the rate of convergence of calculations dynamically so that nonconvergent calculations

can be discovered programmatically. Calculations that iterate to a solution typically assume an initial transient phase characterized by large fluctuations. After a number of iterations that depends on the nature of the calculation, the transient fluctuations die away and the calculation enters a phase in which it converges uniformly to a solution. The difficulty in determining the rate of convergence lies in the problem of differentiating the transient fluctuations from divergence. When it is possible to bound the number of iterations that are subject to transient fluctuations of large magnitude, it becomes a simple matter of computing the rate convergence after the transients are known to have died away. Calculations that do not lend themselves to this method must be treated in other ways. A simple expedient is to fix an absolute upper bound on the number of iterations that can be performed.

4. Recovery from Detected Numerical Computation Failures

In this section it is assumed that the two types of detected failures are loss of significance or excessively poor rate of convergence of a calculation.

Failures involving the loss of significance can be eliminated by recomputation of numerical quantities in multiple precision. Problems for which double precision is insufficient can be eliminated by triple or quadruple precision calculations. Pathological cases do exist, however, that require calculations of even greater precision, but precision requirements for these calculations can be reduced significantly by preconditioning and scaling the data. Recovery from loss of significance should call for recalculation using progressively greater precision until either a significant result is obtained, or the cost of continuing the recalculation exceeds the potential usefulness of obtaining increased significance in the result.

Since the recalculation with increased significance should be invoked automatically by the programming system, we shall consider how this might be effected. There are at least three different methods for accomplishing this.

The first method is based on the control of precision through the program instruction repertoire. Multiple precisions for calculations of varying precision can be prewritten or compiled on the spot from a stored high-level language description of the algorithm. In either case, increased precision is obtained by using multiple precision instructions in the machine instruction repertoire or by making explicit calls on multiple precision software routines.

A second method is to make the precision of the calculation data dependent. Each datum is tagged with a field that indicates its precision. Upon execution of an arithmetic instruction, the lengths of the operands and the lengths of the result of the operation is a function of the tags of the operands. (The Burroughs' B-6500 uses this mode of operation.)

The third method is to make use of microprogrammed, primitive sequences of instructions to implement arithmetic commands, and to use a microprogram memory that is modifiable. The precision of the arithmetic operations can be altered by making changes in the microprogram sequences for these instructions.

Each of these three methods requires further study in context to determine how to handle problems of memory allocation of data so that change of precision can occur with relative ease. Note that in each of these systems, constants must be stored in the greatest precision that might take part in a calculation.

C. Failures Arising from Program Faults

Programs can be extremely complex entities, sometimes far more complex than the computer system on which they are executed. In spite of careful preparation and extensive checkout, rarely do complex programming systems contain no faults. There are several reasons for this. First, programming a complex system is a difficult task. Every contingency must be foreseen and explicit instructions must be written to handle each contingency. The larger systems require a cooperative effort of a large number of individuals, but the demands are such that

the output of each individual must mesh perfectly into the programming system. When systems are tested, their behavior depends not only on the input data, but on the internal state of the system, and quite frequently depends on random timing of events. Exhaustive testing of programming systems over all possible internal state configurations, representative input data, and typical timing conditions is not feasible. For complex systems, even after several continuous months of testing, only an infinitesimally small fraction of possible conditions can be tested.

Hence, to create fault-free programs, several techniques that do not involve exhaustive testing must be used. In this section we describe several techniques for the prevention, detection, and recovery from program faults.

1. Prevention of Programming Faults

There is no panacea for preventing programming faults. It is the nature of programming that every detail must be specified, either explicitly or implicitly. Any single incorrect detail in the program can lead to its failure, and the total volume of detail easily exceeds the volume that a normal human can comfortably focus attention upon at one time. In this section we explore a body of techniques that aid the human in describing the details that constitute a program, checkingout the program, and maintaining that program after checkout.

a. High-Level Languages

The value of high-level languages is well known. They relieve the programmer of a great deal of the burden of specification, and thereby eliminate an important source of error. As the understanding of application programs progresses, the compilers for application programs likewise evolve to include greater power and flexibility. The original FORTRAN compiler gave the programmer primarily the facility for algebraic manipulation, control of loops, definition of data arrays, and formatted input/output. High-level processes have evolved so that recent compilers include processes for memory management, file control

and manipulation, sorting and searching of data bases, and symbolic manipulation of data structures. Languages will continue to evolve, and as they include more complex processes as basic language elements, the problems of writing programs that duplicate these processes will virtually disappear.

The future is not as rosy as the previous paragraph may indicate. The evolution of high-level languages has seen one level of programming errors disappear to be replaced by totally new programming errors. Today's programmers need never write out the details for an addressing polynomial, but they must be capable of determining the proper controls for system executive, language compiler, linkage editor, and relocating loader. Program checkout has actually increased in difficulty because high-level languages let the programmer create more complex programs than might be attempted without their aid, thereby greatly increasing the number of operations within the program that might be faulty. Evolution of programming languages will undoubtedly continue in the current direction of increased facility and power of expression, but the claim here is that languages have tended to overlook the need for improving their reliability.

FORTRAN illustrates the case in point. The language is so defined that programmers need not declare variables. The first reference to an otherwise undeclared variable constitutes a declaration by default. This relieves the programmer of a small burden in return for increasing the unreliability of the language. Every misspelled name of a variable constitutes a default declaration of a new variable so that misspellings cannot be detected by the compiler. On the other hand, the programmer finds that he must declare a large fraction of his variables anyway with DIMENSION, COMMON, and EQUIVALENCE statements so that the small service of default declaration comes at a great cost of reliability.

There are several means for increasing the reliability of a programming language. The most important is to incorporate redundancy into the language so that compilers can perform consistency checks on

statements in the language. Declarations constitute one form of redundancy. Given declarations, compilers can check such things as the type and precision of all variables to determine if they satisfy the conditions imposed by context, the number of dimensions on references to subscripted variables, the agreement of actual and formal parameters of subprograms both in number and type, etc. Formal languages have tended to avoid redundancy instead of embracing it. Yet redundancy is a vital part of natural language, and is essential for communication between humans. The problem than is to investigate how programming languages can be made redundant to increase their reliability without placing undue burden on the programmer.

One answer along this line of thought is to allow the programmer to declare contexts for each variable name. The compiler could check each executable statement to determine if the statement operands can appear in the same local context. Another method is to specify the processes that can alter a variable or may read its value and program the compiler to check references against this specification. These suggestions concern primarily the detection of errors during compilation. Redundancy in the language also gives facility for detecting errors at execution time that cannot be detected during compilation. One such error, for example, is an out-of-bounds index to an array. The remarks here are intended to point out the problem and indicate a direction to take for its solution. A deeper treatment is beyond the scope of this memo.

There are other features of a language that affect its reliability. Some language constructs are error-prone and can be replaced by equivalent constructs that do not lend themselves to error. An example of an error-prone feature in FORTRAN is the specification of Hollerith data using the "nH" format where the n preceding the "H" is a decimal digit that specifies the number of literal characters in the Hollerith string. It has proved to be difficult for programmers to count characters in a string accurately, particularly if the string terminates in blank characters. Later FORTRAN compilers have permitted strings to be delimited at both ends by the "$" symbol and have eliminated the need to count the number of characters in the string. From this example it is suggested that a study be undertaken

to determine what general features of high-level languages are unnecessarily error-prone. What alternatives exist to replace these language elements with equivalent but less error-prone features? An interesting subject for this study is the role of default options of the language. It is conjectured that ill-considered default options share a large responsibility for the unreliability of languages. For example, PL/I default options lead to the unusual side effect that execution of the statement

$$I = 25 + 1/3$$

yields a value for I of 5.333333.

b. Independent Check Calculations

The reliability attained through the use of high-level languages is achieved because the language processors can mechanically generate correct sequences of instructions from terse statements in the language, and because the processors can detect those blunders and transcription errors that lead to inconsistencies in the high-level program. Many faults can escape detection by a compiler because there are elements of processes that must be specified in part by the programmer, and the faulty specification is internally consistent with grammatical and semantic rules of the language. To detect this class of faults, it is necessary to rely on other approaches.

One of the most useful methods of checking for faults is to incorporate independent check calculations into a program. The nature of the check calculations should be to check the functional behavior of program modules in a manner that is not dependent on the internal structure of the modules. For example, a matrix inversion can be checked by matrix multiplication regardless of the particular algorithm that was used to compute the inverse. Check calculations should be done so that they do not merely check functional modules against module specifications because the specifications may have suffered transcription errors. The modules should be checked in their system to see if context meets the

performance requirements of the system. For example, consistency checks on a guidance computer should not merely determine if the guidance computer solves a set of trajectory equations, but rather if the guidance computer solves the set of equations that correctly model the behavior of the space vehicle that carries the computer. Thus programmed consistency checks for this example would compare measurements of the actual vehicle trajectory to the calculated trajectory in order to determine the correctness of the guidance computation.

Consistency checks can be performed at several levels within a programming system. At the lowest level, checks can be performed for many primitive machine operations such as arithmetic and logical operations. When the operation is invertable, it can be checked exactly. When not, it is possible to check if the results of the operation are consistent. Checks at this level will not detect programming faults per se, but rather detect hardware failures that effect particular primitive operations.

Consistency checks of higher-level processes can be performed in much the same way as for lower-level processes. When processes have inverses, the consistency check can be the inverse process. For example, root extraction from polynomial equations can be checked by evaluating the polynomial with the extracted root as an argument. Processes without inverses can be checked by determining if the output values are self-consistent and consistent with the input values and initial state of the process. As an example of a checkable process of this type, consider the programmed model of a physical process undergoing a smoothly fluctuating change of state while under the influence of smoothly fluctuating inputs. If a sharp discontinuity is discovered in the output of the process, then it is likely that there is a fault in the program, or else the discontinuity is characteristic of the process, and its detection could have been predicted before hand. Thus, monitoring the output of the process for discontinuities is a satisfactory check.

To ensure ultrareliability of a program, all processes should be checked at some level, possibly at several different levels by several different checks. Among the factors that affect the placement and number of consistency checks are:

(1) The cost of performing the check calculation in terms of programming effort, execution time relative to the execution time of the process that it checks, and total system memory requirements.

(2) The cost of experiencing an undetected failure in the module that is checked.

(3) The probability that a failure can escape detection if a check calculation is performed.

(4) The resolution of the check calculation in terms of its capability of determining the location of a failure.

(5) The probability that the check calculation contains a fault.

Of course, during the initial phases of program checkout, a great many checks should be performed in order to obtain high resolution of the location of faults causing detected failures. This entire discussion is directed to the final phases of checkout and full operational status when it is desirable to execute without the overhead of elaborate check calculations, yet maintain a certain level of reliability.

The problem of determining how many check calculations to perform and where to place them is the software equivalent of the classic problem of designing redundant hardware. Software redundancy, like hardware redundancy, has a definite place in an ultrareliable computer system. It should be designed into system from the initial inception of the system. The form and degree of redundancy that should be put into software is a function of factors particular to the situation. As this subject is studied further, perhaps there will emerge some general methods for building redundant software as has happened for hardware.

c. Software Maintenance and Modification

The inherent flexibility of software with respect to the fixed structure of hardware usually results in system modifications being thrust upon programming systems rather than upon the hardware that supports the programming system. The cost of these modifications is quite high in terms of reliability because changes can cause new errors to appear in systems that are otherwise error free, and the new errors may escape detection during system checkout. There are two problems to be faced here. How the programs are to be written to minimize the possibility of introducing new errors when they are modified, and how newly modified systems are to be tested to determine if new errors have appeared. In this section we consider several techniques for attacking these problems.

An important factor in the solution to the problem of software modification is in the design of the software system. Software should be designed to accommodate changes easily, as if changes are unavoidable. In most cases, changes are truly unavoidable because it is difficult to anticipate all of the required characteristics of a software system until it has been constructed.

A good method for accommodating changes easily is to design modular software systems. Subprograms should be organized functionally so that they operate independently, communicating only through a minimum set of parameters. System constants should be treated as parameters so that the modification of a parameter in a single point in the program causes all references to that parameter to be altered. A process that is common to several subprograms should be factored into a module that is called by the several subprograms. For languages like ALGOL and FORTRAN, factoring is essentially equivalent to the construction of a subroutine, block or overlay.

Factoring of processes may not require that the processes be physically partitioned, as is the case when processes are factored into subroutines. The modularity required for ease of program maintenance is semantic modularity, and this is a by-product of physical modularity. Physical partitioning produces a certain amount of program inefficiency

because of the overhead for establishing linkages during program execution. Semantic modularity can be attained without physical partitioning and its attendant inefficiency, by using the programming device known as macro-expansion. Macros have been available for many years in low-level assembly languages and are only recently being used in high-level languages. To date, the implementation of macros in high-level languages has been somewhat limited, but it appears that macros are a natural adjunct to common algorithmic languages. Undoubtedly, further effort will be given to increasing the power and utility of macros in high-level languages, so that it will become increasingly easier to modularize programs without loss of efficiency.

It is possible to carry the partitioning process too far so that the net result is a loss of reliability rather than an increase. This occurs when similar but nonidentical processes are grouped in a common module. Usually this is done by including several tests to differentiate among the various processes at appropriate points in the module. The problem with this type of partitioning is that a modification to one process can affect all other processes that share the same module.

The problem then remains one of determining how and when to modularize. Should the programmer use a macro or subroutine? Should two processes be placed in separate modules if they differ in only one respect? If not, then how different should they be in order to be separated? To answer these questions, the programmer must apply his experience and judgement and consider the factors of the particular situation. That fact that only subjective characteristics are considered here is indicative that program partitioning is still very much an art. Its importance has been recognized to the extent that aids for partitioning are common in high-level languages. These include block structures, multiple job-step programs, program segments, overlays, and asynchronous tasking. There is still much to be done to aid modularization so that standard software packages can be written, completely checked, and debugged, then used in many systems. This requires that

standards be established to guarantee standard program interfaces, file structures, and languages.

A second important factor affecting the maintenance of software is the documentation of the software system. A problem in developing adequate documentation currently exists, because it has not been firmly established what constitutes adequate documentation. Several methods have been commonly used, none of which has been entirely satisfactory. For our discussion here we distinguish between two types of documentation, primary and secondary. The primary document is the source listing of the program. It is the document that can be mechanically translated into an executable program. Secondary documentation is any material that describes the program but cannot be mechanically translated into an executable program.

It is generally conceded that primary documentation, as we know it today, is inadequate for maintaining and updating complex software systems. The reason being that the requirements for describing a process so that a computer can perform the process are quite different from those for describing the same process to a human so that he can gain an understanding of it. The role of program documentation is to state both the intent and the methodology of a program, and to describe the structure and meaning of the program data. The primary document describes the methodology but not necessarily the intent of the program, and gives the structure of the data but not necessarily its meaning. Thus, the primary document may not be sufficient documentation, even though it is a complete description of the executable program.

Through the use of symbolic programming languages, it is possible to enhance the descriptive qualities of the source document to make clear the intent and meaning of a program. The programmer can use descriptive names and phrases for program elements and data structures so that the source document takes on the qualities of a narrative description. Consider, for example, the mnemonic content of the name INPUT as the name of a subroutine and how much more descriptive that name is compared to a name like QSIA, which might be equally acceptable as a

symbolic name. Partitioning can also be used to enhance the descriptive nature of the primary document. Program details often tend to decrease the comprehensibility of the program. When a set of steps are factored into a program module, and that module is given a descriptive name, the effect is to replace a lengthy portion of relatively uniformative text with a short terse description of the intent of the text. Past experience has shown that, in spite of the descriptive qualities of symbolic languages, there must still be additional documentation to aid program maintenance. Therefore, secondary aids have been used heavily to document systems.

One of the shortcomings of primary documentation is the difficulty in following the flow of control in a program. A human cannot easily trace program branches, because it is difficult to relate the source of a jump to its target when the two are physically remote in a listing. Consequently, the flowchart has gained a good deal of popularity because of its ability to make these linkages visible. But flowcharts do not fully satisfy the need for documentation either. They clarify the description of the methodology, but they give neither the intent of the program, nor do they give meaning to the data. They fail to give an adequate presentation of intraprogram communication, when the communication is done through common data rather than through the flow of control. Programs that make use of recursive processes or asynchronous tasks do not lend themselves well to descriptions by flowcharts. Finally, because flowcharting is a secondary aid, the flowcharts must be maintained and updated as the programs that they describe are updated. When the flowcharts are done by hand, the updating is costly in terms of utilization of human resources. Automatic flowcharting has become available recently, thereby eliminating some of the problems of flowchart maintenance, but it still does not remedy the other problems mentioned above.

To a large extent, the high-level languages have overcome the difficulty in displaying the flow of control in a program. By avoiding the "GO TO" construct and making use of phrases such as "IF...THEN...ELSE" or one of the common iterative control statements, the programmer can

increase the readability of his source document. Partitioning can also be used to keep the flow of control in a program in a narrow context. Thus, when properly used, high-level languages come close to flowcharts with respect to their capability for displaying the flow of control.

Machine produced cross-references are commonly used to document the interaction of data and program modules. Cross references often contain information as to the context of each reference to a variable name. They might indicate whether the variable is read or written, or passed as an argument to a subprogram, or if the reference is in a comment or declaration. This information is extremely useful when modifications are made because the effect of changing a single variable can be traced to all pertinent contexts in the program. Information of this type is not present in flowcharts or in the source program itself.

An important type of documentation known as the decision tables has grown popular as a replacement for the flowchart. Decision tables are two-dimensional tables in which a list of processes is arranged along the row axis and a list of conditions arranged along the column axis. An entry at a row-column intersection in the table indicates that the row process is to be executed when the column conditions are satisfied. Usually the columns represent a set of mutually exclusive conditions, and multiple entries in a single column are assumed to be executed sequentially from top to bottom of the table. The value of decision tables is that they show the flow of control of high-order processes, and, hence, lend insight into the intent of the program. Mechanically generated flowcharts tend to show more detail than necessary, and thereby tend to hide the intent of the program.

A novel aspect of decision tables is that it is possible to use them as a primary source rather than as a secondary source of documentation. That is, the tables constitute a high-level programming language that can be translated into executable form. Thus, decision tables are a form of self-documenting program.

An interesting problem that might be treated in the future is the problem of designing programming languages and techniques that extend the notion of self-documenting program. Certainly, one step is to provide higher-level operations in the source language. The COBOL SORT verb exemplifies this methodology. Another technique is to permit and encourage the programmer to use descriptive names. FORTRAN is overly restrictive in this sense, because identifiers must be six characters or less. The length restriction could be doubled or tripled without over burdening the compiler and provide the programmer with a much greater descriptive capability. Other techniques that might be considered are natural language processing, and the use of graphic languages with display input-output.

The second major topic to be considered in this section is the problem of testing a program during program checkout. How does one go about determining the correctness of a program? The most common methodology is to check each program module independently, then to check increasingly larger collections of modules until the full system is tested. Although this is a good methodology, it is insufficient by itself because many errors can escape detection unless a more methodical approach is taken.

Another technique is to generate test data such that it is guaranteed that every branch of the program is executed at least once. An interesting problem associated with this technique is the automatic generation of test data by the source language compiler to test every branch completely. Another technique that might prove valuable during checkout, when the true behavior of the program is not known, is to mark program instructions as they are executed in order to identify those instructions that have not been exercised. As checkout proceeds, the list of unexercised instructions can be used to guide the latter phases of checkout, and to ensure a checkout completion that every step has been tested for at least one set of data.

Even the type of test described in the previous paragraph is insufficient, because the behavior of a program is a function of both the input and the state of the program. Hence, the program may accept

a set of data and operate correctly during one iteration and fail during the second iteration on the very same set of data. Assuming that an exhaustive test of the program is not feasible, a good approach is to design a thorough test that takes into consideration the state of the program. To do this, the input data is partitioned into several possible classes and similarly the possible program states are divided into several different classes. Given both the classes of input and the program states, one then designs a set of test data that will test the behavior of the program for every pair of input data-program state classes. The thoroughness of the check depends on the fineness of the state and input class partition.

When modifications are made to programming systems, the common practice of checking only the changes that were made to the system often leads to undetected errors in the program. The problem here is that if an error is introduced into the system, it can occur because of subtle interactions that are connected only remotely to the sections of the program that were changed. Furthermore, if the interactions had been foreseen, while the changes were being made, they might have been avoided. When the system is checked only for those cases which are known to be affected, the subtle problems may not appear. It is important to recognize, therefore that a change to a program produces a completely new program that must be treated as if it were completely untested.

2. Summary

In this section we have considered a large number of problems related to writing and checking out programs. The single most promising approach for achieving ultra-reliable programs appears to be in the use of high-level languages. Through high-level languages, it may be possible to mechanically generate the check calculations, test data, and program documentation that have been described elsewhere in this chapter. In effect, the computer is the ideal tool for aiding the programmer in developing a reliable program.

D. Techniques for Detecting Software Failures

In this section we assume that failures might occur in programming systems in spite of careful program preparation and extensive system checkout. Program faults might escape detection during checkout, but occur during normal system operation if for some reason the conditions under which the failure can occur are satisfied. Those transient hardware failures that have effects that are similar to software failures can also be detected by ultrareliable programming techniques. The purpose of the techniques that we describe in this section is to protect the system from software or transient hardware failures by detecting failures at the earliest possible time and returning information that enables the system to recover from the failure. Additional information that is returned might be used to diagnose the possible cause of the failure. We stress software techniques in this section, but hardware techniques are considered where pertinent.

1. Protection Against Incorrect Memory Accesses

One of the least reliable aspects of computer systems is the volatility of the memory. When a failure causes information in memory to be destroyed, it may not be possible to recover the lost information; thus, the failure is an irreversible action. It is unfortunate that a characteristic of program faults is that they cause incorrect memory operations to occur, with the attendant loss of information. A faulty memory operation may be one of two kinds. Either good data is replaced by incorrect data, or more seriously, the address of a memory operation is incorrectly computed, thereby causing the storage of good data in the wrong place in memory.

A fault that causes information to be lost may or may not be a serious one in terms of the survivability of the programming system. Clearly some information is essential for the survivability of the system. Programs must not be altered by faults, for example, nor can data that holds linkages and status information. Therefore, we shall assume that a class of critical information can be identified, and that this class can be given special protection.

The most serious type of failure is the write operation, because it is inherently destructive. However, for very little extra cost, it is possible to protect both read and write operations and gain additional reliability.

When critical information is constant, it can be placed in a read-only memory to assure its nonvolatility. It might be "wired-in," or stored in a memory that has both a nondestructive and destructive read-out capability. Once the data is stored, the assumption is that the memory is switched to nondestructive mode. Privileged subprograms might be given the capability to alter the contents of the protected memory, but the memory is always placed in the nondestructive mode after modification. We note here that it is possible to use conventional memories together with special hardware in the access circuitry to implement protection against unauthorized attempts to modify critical data. In a sense, the conventional memories can be made to simulate memories with both destructive and nondestructive read-out, and a protection capability. This type of system has been implemented in several commercial computers.

The protected memory is most effective in protecting against faults in nonprivileged programs. Privileged programs can be faulty also, so that there must be some protection against faults that successfully penetrate the memory protection. The most important aspect of assuring reliability in this case, is to be certain that critical data can be restored if it is lost. One method for doing this is to dump the contents of critical memory periodically, and restore the memory from the last checkpoint when a fault is detected. Another method is to retain the last few values for each item in critical memory so that values can be restored selectively. Care must be used in the implementation of the latter technique because data cannot, in general, be altered out of context. It is necessary to change sets of data so that all values have meaning as a collection in the context of the programming system.

The technique of using a real or simulated read-only memory is valuable for protection of noncritical data, and for protection of the programs that manipulate these data. Suppose that all of memory can be partitioned into blocks that are given either read-write or read-only status. We shall assume that the status can be altered dynamically by some programming mechanism. Then invalid addresses that are generated may be detectable when they lead to attempts to write destructively in a read-only block. Programs can always be placed in read-only status when they are written as reentrant programs, i.e., they do not modify themselves. Data structures might be capable of being in either status. A valuable feature in a reliable programming system is the ability to set and alter the status of data structures as required. The techniques can be extended further by considering other states that might characterize blocks of memory. For example, execute-only is a state that could protect machine instructions from being read erroneously as data, or could also prevent data from being erroneously treated as instructions.

The ability to define protected states places some demands on both the hardware and the software. The state information must be held at some point in memory from where it can be retrieved quickly during each memory cycle. It might be held in a register or in a special memory plane. It might be stored in a tag field that is associated with every word in memory so that it is available for checking during any read operation or write following a destructive read operation. In any case, it is clear that the status of a memory block must be matched against the type of memory request to determine if an illegal operation has occurred.

Several hardware features would be useful in supporting the memory protection techniques mentioned above. Reentrant programs benefit from instruction sets that include special instructions for facilitating reentrant coding. For example, the common practice of depositing subroutine addresses at fixed linkage points within the body of a program violates the rule that reentrant programs must be read-only. To aid reentry, the return link should be stored in a register or in a data

area that is accessed indirectly through a linkage address. To aid the protection of the critical data area, ultrareliable hardware designs should be utilized in the block of memory that holds critical data. This lowers the probability that critical data will be lost because of hardware failures. Memory access circuitry should be designed redundantly to minimize the probability that valid memory requests will be honored at invalid addresses. Paging hardware and relocation registers are useful items for storing status information of memory blocks. Status checking is a natural extension of the processing that occurs with paging and relocation hardware during the computation of effective addresses.

The main point of this section is to bring to light techniques for protecting against invalid memory requests. In the next section, we consider techniques for broadening the scope of the protection.

2. If and Only If Programming

The previous material on the protection of memory operations is an important example of the notion of if and only if programming. Memory protection is possible because the program and data essentially defines the region of memory that is addressable at any given time. But it also defines the region of memory that is not addressable. Furthermore, different portions of a program may have different addressable and unaddressable regions. The protective measures generally compute for each effective address whether it is in a region that is accessible or unaccessible by the process requesting a memory operation.

Thus a request for a memory access is honored:

(1) if a process issues the request, and

(2) only if the request is valid.

Carrying this notion further, we postulate that programs should be analyzed to determine not only what combination of input data and internal states are valid, but to determine those combinations that are invalid. When an invalid combination occurs, a failure has occurred, which should trigger a detection mechanism.

As another illustration of the notion of if and only if programming, suppose that a program is tested thoroughly, and it is determined that the program is completely correct. It may be the case, however, that some variable ranged only through the values between 0 and 10 during the checkout phase, and it is known beforehand that no other values can be encountered during normal operation. What happens to the program if the variable in question takes on a negative value? By assumption, this cannot occur in normal circumstances, but suppose the negative value were introduced by an undetected failure. The behavior of the program is not defined, and its true behavior in this circumstance can be anything from harmless to disastrous.

The use of if and only if programming in this example would begin with the determination that the true range of validation of the input variable is the range 0 to 10, and would follow with the insertion of a test of the variable to determine if it lies in the range of validation.

One class of items that are easily checked are the class of variables that are used as indices of elements of a data structure. When a variable is an index into a particular data structure, then the valid range of values for that variable is completely determined by the data structure. The variable should be checked before every instance in which it is used to address an element of the data structure. This type of check in one sense is comparable to the memory checks described in the previous section, but the following difference is observed. The memory checks described previously are made after a memory request is issued, and are made on the basis of status of the memory at the effective address and the type of memory request. The check of the index variable is made before a request is issued, and is made on the basis of a variable that is used to calculate an effective address, rather than on the nature of the data found at the effective address.

Indices to data structures can be checked by range checks to determine if the index is within the actual bounds of the structure. In commercial systems, bound checks of this type are common, and have been implemented by both hardware and software techniques. The techniques generally postulate that the bounds of a structure are associated with

the base address of the structure, so that whenever the base address is used to calculate the true address of an element of the structure, the bounds are also available to be used in the calculation of range checks.

The technique mentioned above is adequate for some types of data, but is not sufficient to protect physically large data structures when there is a high probability that any incorrect address lies within bounds. One class of data that satisfies this criteria is the linked-list data structure. Most of the data within linked-lists are addresses (or indices) of other elements in the list-structure. Within the list structure, the linkages may identify several different classes of data, but the linkages usually have one form that is common to the entire structure. Faults that generate incorrect linkages might be undetectable by bounds calculations. A better check is to store with each pointer the class of the item to which it must point. With each item a field is used to indicate its class. Whenever a pointer is used to access an item, the class of the fetched item is compared to the class that was expected, and an error is signalled if they do not agree. A problem arises when a pointer might point to one of several classes. The check operation then becomes rather lengthy and contributes system inefficiency. A variation of this technique eliminates the problem of overhead. Instead of associating classes with data, each item is associated with a randomly selected tag. When pointers to the item are created, a copy of the tag is placed in each pointer. Whenever an indirect access is made through a pointer, the tag of the pointer must agree with the tag found with the item addressed.

Indices and address linkages are just one class of variables that are easily checked by the if and only if technique. In general, it is possible to perform a great variety of checks on data, and possibly on a large majority of the variables in a program. When a variable is associated with a physical process, then it is highly improbable that the variable would take on values throughout the range of numbers within the range of representation. Hence, the variable can be tested periodically to see if it falls outside a realistic range of values.

Endless loops can be prevented in iterative programs by using roughly the same technique. Every iterative segment is given a unique loop counter to keep track of the number of loop iterations after initial entry. The loop counter then is a variable whose range of values is bounded by the largest number of iterations that might have to be executed to complete processing. Thus, a bound can be introduced for every loop counter (it need not be a least upper bound), and the loop counter becomes a special case of the type described above.

The examples given here suffice to illustrate the application of if and only if checks. The general principle is to make use of information about the physical significance of data or internal program variables to determine their valid range of values. Then checks should be inserted into programs at appropriate points to ascertain that the values stay within the ranges determined.

3. Recovery from Detected Faults

There are two problems that must be treated before bringing the problem of error detection to a close. The first problem is to determine what to do after an error is detected in order to maintain the viability of the programming system. The second problem is to return as much information as possible that will aid in the diagnosis of the detected error. Since these problems are related, we treat them both together.

It is assumed that there will be a hierarchy of programs in the programming system with at least two levels in the hierarchy. The highest-level program is the system executive, while the application programs occupy the lower levels. It is natural that detected errors cause control to be reverted either directly to the system executive or to the program that is one level higher than the level at which the error occurred. Both types of error exits may exist within one system, and the type of detected error could determine the point at which control is to resume. It is clear that every program that might be the target of an error exit should have a subsegment defined that will be the point of resumption of control. Furthermore, the address of the error entry should be posted in a linkage area that is immediately available to the

hardware in case an error is detected. In this way, a recovery procedure will be available for every possible detected error.

A problem exists when a detected error occurs for programs at the highest level. Where should control be transferred in this case? Clearly, if the wrong transfer is taken, the error may be repeated, and thereby begin an infinite loop. There is no good answer for this problem. It may be necessary to attempt recovery in a standard fashion and simultaneously notify a human operator or an external computer of the nature of the problem. A technique that will tend to reduce the magnitude of the problem is to make the highest-level program as small as possible to reduce the probability of making and detecting errors at that level. The common practice of using two-level hierarchies places a large burden on the executive programs, and thereby increases the probability of faults occurring while operating in executive mode. The use of a third level for the highest level would be far more satisfactory, because most executive processes could be second level processes and receive a greater degree of protection than can be obtained from two level systems.

Assuming that recovery points for every type of detectable error can be defined properly, except possibly for those errors that are detected in the top level programs, the problem remains one of diagnosing the reason for the detected error. It is essential that the context of the processor at the time of the error be saved and be made available for diagnostic purposes. The context is the contents of all visible registers, status bits, the instruction counter, the effective address of the last memory access, and other pertinent data. If the data is to be returned to a human, then it should be analyzed and translated into terms that will ease the problem of interpreting the data. Symbolic names can be supplied from symbol tables that might be stored in auxiliary memory for diagnostic purposes. Linkage traces can be used to determine the return linkages of subroutine calls. Arguments of subroutines and the values of important variables can be posted to give further diagnostic aid. Since program jumps usually prevent program steps from being traced backwards, it is recommended that jumps leave a return address in a special register or

set of registers organized as a small stack. At the point of error detection, the register or register stack will contain the source points of the last few branches.

It is recommended that no matter how severe the detected error, the system should maintain integrity for a period that is sufficiently long to gather and output diagnostics. The reason being that the fault that caused the failure may lead to failures under rare conditions that are difficult to analyze or repeat. When this type of failure returns diagnostics, insufficient for correction of the fault, then it is highly probable that it will remain in the system indefinitely.

E. Summary and Conclusions

In the course of this discussion, several aspects of the reliability of programming systems have been explored. We have treated the subject from a rather general view-point, primarily attempting to determine what techniques are generally applicable and problems remain unsolved. If we try to draw a parallel with the design of reliable hardware, then software lacks a theory of reliable construction that is enjoyed by hardware. In terms of practical designs, both hardware and software share a black art for the design of reliable systems, where the experience and judgement of the designer ultimately determines the reliability of the system.

If a theory of reliable software is to emerge, it is most likely to be predicated on the use of high-level languages to describe manipulations in terms of thoroughly checked standard packages. Redundancy of expression will be used to automate the insertion of check calculations and recovery routines. Certain types of errors will be avoided, because there will be no way of expressing them in the high-level languages.

Several interesting problems have emerged from the preceding discussion. These are collected here in the hopes of stimulating further interest in the area. The problems are:

1. What methods for automatically changing the precision of a calculation?
2. What features of languages are error-prone? How might they be eliminated in favor of equally powerful but less error-prone features?
3. How can redundancy be utilized in programming languages to protect programmers from errors?
4. Is it possible to develop an abstract model of a program and use the model to determine the placement of check calculations? Can the model be used to guide the generations and placement of check calculations for practical programs?
5. What is adequate program documentation? How can languages alleviate the problem of documentation by becoming more self-documentary?
6. Can updating be computer assisted to eliminate the propensity for introducing new errors by updating?
7. How can programs be designed to be modular? What standards are necessary to aid modularity?
8. How can tests be generated or built into the hardware to give extensive protection for a small decrease of efficiency?
9. How can recovery be made from errors detected in the "hard-core" software? What diagnostics should be returned after any detected failure and how should they be returned? Can special routines perform diagnosis?

VI CONCLUSIONS AND SUMMARY OF OTHER STUDIES IN PROGRESS

In this chapter we briefly present our conclusions on techniques for the realization of ultrareliable spaceborne computers. These are based upon both the research conducted during the second phase and prior related work, and also recommendations for the future direction of research in this area. We also summarize other studies that have either not yet progressed to a state where reporting is appropriate or do not bear directly on the technical contents of this report.

A. Conclusions

(1) The computer reliability requirements of an advanced spaceborne mission cannot be satisfied without the use of redundant logic structures.

(2) Practically any reliability constraint can be satisfied by the exclusive use of passive masking techniques. But, with the exception of mission tasks of relatively minimal complexity, the cost of such a computer would be excessive. Significantly improved utilization of resources is theoretically achieved with a reconfiguration technique in which the logical interconnections can be altered so that faulty units are disconnected from the system and moreover, with a graceful degradation technique in which the scheduling of tasks can be altered to match the available performance capability.

(3) For advanced spaceborne missions there exists, in addition to the severe reliability constraint, the requirement to accommodate to simultaneous introduction of several problems and to varying measures among the mission problems of priority, accuracy, and urgency. The multiprocessor framework appears to be the best match to these requirements.

(4) The reliability of the multiprocessor is significantly enhanced if a limited degree of fault-masking and reconfiguration (or repair) capability is incorporated within the memory, control, and processor modules.

(5) The major problems associated with this repairable multiprocessor are the design of memory, control and processor modules, which are either amenable to repair or fault masking; the design of commutation networks for the data switching; the specification of diagnostic techniques for the detection and location of faults; and the overall reliability analysis of the system.

(6) Processor modules, which include microprogram control and which are amenable to repair, can be designed by organizing the logic associated with each byte or computation into a slice or module of moderate complexity (i.e., approximately 1000 gates/module). The repair operation is then the electrical shorting of data around a faulty slice.

(7) Commutation networks can be designed which are suitable for the routing of data between memory and processor modules and also for the above defined repair operation. These networks can be easily set up and diagnosed, and can be made insensitive to failures within the commutation network with a moderate increase in complexity.

(8) It appears feasible to synthesize programs that can detect, utilizing combinations of software and hardware redundancy, the occurrence of many hardware faults. These techniques are also appropriate to the specification of formal rules for the synthesis of programs that do not contain human mistakes.

B. Summary of Other Work in Progress

(1) Work is continuing on the description of logical design techniques for the various module types of the multiprocessor. During the next period, a substantial effort will be devoted toward the design of the irregularly-structured control module, and in particular, to the investigation of the optimum balancing of the various reliability-enhancement schemes.

(2) Work will be initiated on diagnostic techniques for locating the block within a module that is suspected of being faulty. We expect to consider the diagnosis of byte-sliced processor modules, commutation networks, associative memories of the type used for memory re-addressing and the table of available equipment, and general control logic. We have looked at the possibility of including auxiliary outputs to facilitate diagnosis, and formulated the problem of specifying the optimum set of such outputs as a covering problem of the type similar to that described in Sec. III-B of Ref. 1.

(3) Work has begun on the analysis of the multiprocessor system, and we are seeking models which are amenable to analysis.

(4) The survey of the literature pertinent to the problem of improving reliability by the use of redundant structures has been completed, and a paper summarizing this work will be submitted for publication to one of the IEEE Transactions.

Appendix

USE OF CODES FOR CORRECTION AND DETECTION

It is of interest to determine the probability of an undetected error due to a coherent noise signal that affects all channels.

The number of detectable errors for a double-error correcting code may be computed as follows:

Let

n = the number of bits in the code word

k = the number of non-redundant data bits;

then the combined number of valid and corrected patterns is

$$2^k\left[\binom{n}{0} + \binom{n}{1} + \binom{n}{2}\right] = 2^k(n^2 + n + 2)/2 ,$$

out of 2^n total patterns. The fraction of error patterns that are only detected is thus

$$f_d = 1 - \frac{2^k(n^2 + n + 2)}{2^{n+1}} ,$$

or approximately,

$$1 - \frac{n^2}{2^{n-k+1}} .$$

This fraction is approximately 1/2 for the attractive n = 16, k = 8 code, and increases rapidly for larger word lengths. If B bytes are used, and if all channel errors are independent, the probability of an undetected error is f^B; for example, for the 16,8 code, with B = 3 (i.e., 24 non-redundant bits per word), f^B is 1/8. In the case in which noise transient lasts for several memory cycles (very likely for noise due to

electrical arcing), this probability decreases by a factor of f^B for each cycle.

We conclude that the normal error detection capability of error correcting codes provides reliable warning of the existence of massive errors.

REFERENCES

1. J. Goldberg, K. N. Levitt, and R. A. Short, "Techniques for the Realization of Ultra-Reliable Spaceborne Computers," Final Report-Phase I, Contract NAS 12-33, SRI Project 5580, Stanford Research Institute, Menlo Park, California (September 1966).

2. AES-EPO Staff, "AES-EPO Study Program" Final Study Report, Volumes 1 and 2, IBM Electronics System Center, Owego, New York (December 1965).

3. A. Avizienis, "A Set of Algorithms for a Diagnosable Arithmetic Unit," Tech. Report No. 32-546, Jet Propulsion Laboratory, Pasadena, California (1964).

4. A. Avizienis, "A Design of Fault-Tolerant Computers," Proc. Fall Joint Computer Conference (AFIPS) (1967).

5. W. G. Bouricius, et. al, "Investigations in the Design of an Automatically Repaired Computer," Digest of the First Annual IEEE Computer Conference, IEEE Publication 16C51 (September 1967).

6. P. W. Agnew, et. al, An Approach to Self-Repairing Computer," Digest of the First Annual IEEE Computer Conference, IEEE Publication 16C51 (September 1967).

7. R. P. Hassett, and E. H. Miller, "Multithreading Design of a Reliable Aerospace Computer," presented at 1966 Aerospace and Electronic Systems Convention (3-5 October 1966).

8. L. J. Koczela, "Study of Spaceborne Multiprocessing," 2nd Quarterly Report, Volume II, Contract NAS 12-108, Autonetics Division of North American Aviation, Anaheim, California (October 1966).

9. E. C. Joseph, "Self Repair: Fault Detection and Automatic Reconfigurability," Proceedings of the Spaceborne Multiprocessing Seminar, NASA Electronics Research Center, Boston, pp. 41-49 (31 October 19

10. R. L. Alonso, et. al, "A Multiprocessing Structure," Digest of the First Annual IEEE Computer Conference, IEEE Publication 16C51 (September 1967).

11. J. F. Keeley, et. al, "An Application-Oriented Multiprocessing System," IBM Systems Journal, Volume 6, No. 2 (Entire Issue) (1967).

12. J. J. Pariser, "Multiprocessing with Floating Executive Control," IEEE International Convention Record (1965).

13. S. P. Frankel, "On the Minimum Logical Complexity Required for a General Purpose Computer," IRE Trans. on Electronic Computers, Volume EC-7, No. 4, pp. 282-285 (December 1958).

14. H. Weber, "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," Communication of the ACM, Volume 10, No. 9, pp. 549-558 (September 1967).

15. A. Grasselli, "The Design of Program-Modifiable Microprogrammed Control Units," IEEE Trans. on Electronic Computers, June 1962, pp. 336-339.

16. J. Goldberg, "Logical Design Techniques for Error Control," WESCON paper 9/3, Session 9 (September 1966).

17. W. H. Kautz, K. N. Levitt, and A. Waksman, "Cellular Interconnection Networks," Accepted for publication in IEEE Transactions on Electronic Computers.

18. A. Waksman, "A Permutation Network," Accepted for publication in the Journal of the ACM.

19. W. J. Cody, "The Influence of Machine Design on Numerical Algorithms," AFIPS, Proceedings of the SJCC, Thompson Books, Washington, D.C., pp. 305-310 (1967).

20. Peter M. Neely, "Comparison of Several Algorithms for Computation of Means, Standard Deviations and Correlation Coefficients," Communications of the ACM, Volume 9, No. 7, pp. 497-499 (July 1966).

21. N. Metropolis, and R. L. Ashenhurst, "Basic Operations in an Unnormalized Arithmetic System," IEEETEC, Volume EC-12, No. 5, pp. 896-904 (December 1963).